#2

The Patent Office
Concept House
Cardiff Road
Newport
South Wales
NP10 8QQ

J1002 U.S. PTO
09/940202
08/27/01

I, the undersigned, being an officer duly authorised in accordance with Section 74(1) and (4) of the Deregulation & Contracting Out Act 1994, to sign and issue certificates on behalf of the Comptroller-General, hereby certify that annexed hereto is a true copy of the documents as originally filed in connection with the patent application identified therein.

In accordance with the Patents (Companies Re-registration) Rules 1982, if a company named in this certificate and any accompanying documents has re-registered under the Companies Act 1980 with the same name as that with which it was registered immediately before re-registration save for the substitution as, or inclusion as, the last part of the name of the words "public limited company" or their equivalents in Welsh, references to the name of the company in this certificate and any accompanying documents shall be treated as references to the name with which it is so re-registered.

In accordance with the rules, the words "public limited company" may be replaced by p.l.c., plc, P.L.C. or PLC.

Re-registration under the Companies Act does not constitute a new legal entity but merely subjects the company to certain additional company law rules.

Signed [illegible]

Dated 25 JUN 2001

THE PATENT OFFICE
M
10 APR 2001
NEWPORT

10APR01 E621050-1 C59521
P01/7700 0.00-0108953.1

Patents Form 1/77

Patents Act 1977

The Patent Office

1/77

310002 U.S. PTO
09/940202
08/27/01

Request for grant of a patent
Grant

10 APR 2001

The Patent Office
Concept House
Cardiff Road
Newport
Gwent NP10 8QQ

1.	Your reference	2034-P553-GB
2.	Patent application number	0108953.1 NEW
3.	Full name, address and postcode of the or of each applicant *(underline all surnames)*	DISCREET LOGIC INC 10 Duke Street Montreal Quebec Canada H3C 2L7
	Patents ADP number *(if you know it)*	6972319002
	If the applicant is a corporate body, give the country/state of its incorporation	Quebec, Canada
4.	Title of the invention	INITIALISING MODULES
5.	Name of your agent	ATKINSON BURRINGTON
	"Address for service" in the United Kingdom to which all correspondence should be sent	25-29 President Buildings President Way Sheffield S4 7UR GB
	Telephone No:	0114 275 2400
	Patents ADP number	7807043001 ✓

		Country	Priority application number *(if you know it)*	Date of filing *(day/month/year)*
6.	If you are declaring priority from one or more earlier patent applications, give the country and the date of filing of the or of each of these earlier applications and *(if you know it)* the or each application number	N/A	N/A	N/A

		Number of earlier application	Date of filing (day/*month/year)*
7.	If this application is divided or otherwise derived from an earlier UK application, give the number and the filing date of the earlier application	N/A	N/A

8.	Is a statement of inventorship and of right to grant of a patent required in support of this request?	YES

9. **Enter the number of sheets for any of the following items you are filing with this form. Do not count copies of the same document**

Continuation sheets of this form	
Description	26
Claim(s)	09
Abstract	01
Drawings	20+20

10. If you are also filing any of the following, state how many against each item.

Priority documents	NONE
Translations of priority documents	NONE
Statement of inventorship and right to grant of a patent (*Patents Form 7/77*)	NONE
Request for preliminary examination and search (*Patents Form 9/77*)	NONE
Request for substantive examination (*Patents Form 10/77*)	NONE
Any other documents (*Please specify*)	

11. I/We request the grant of a patent on the basis of this application.

[illegible]

Signature

Date **Friday, 06 April 2001**

12. Name and daytime telephone number of person to contact in the United Kingdom

RALPH ATKINSON CPA
0114 275 2400

SR-04-039

Initialising Modules

Field of the Invention

The present invention relates to initialising a plurality of modules of processing instructions prior to execution of an application running on a computer.

Introduction to the Invention

All processing devices, whether they be personal computers, personal digital assistants or embedded processors, suffer from complexity. The speed and capacity of processing and data storage devices has shifted the limits from hardware to application instructions.

The complexity of many applications is such that it is no longer possible to consider a set of application instructions as ever being finished. Instead, the source code for these instructions evolves over time, improving and following changing user requirements over a product lifetime of many years. Even in the short term, application instructions may change from day to day. In order to deal with this constant evolution, and to minimise the problems of managing complexity, it is established good engineering practice for applications to comprise a large number of small sets of instructions or modules. Individual teams of engineers can work on modules separately, and this permits parallel evolution of several aspects of application behaviour. Theoretically, any complex application can be broken down into sufficiently small individual modules so that complexity, at the module level, never becomes a limiting factor. However, as the number of modules increases, the problem of combining them to work together becomes more difficult. In the art, it is this problem which places an upper limit on the complexity of reliable application evolution.

A particular difficulty when combining modules in an application, is module initialisation, which has to occur before the main application processing begins. In order to avoid this problem, engineers have to keep application complexity as low as possible, while still fulfilling the application requirements.

In certain environments, such as complex power generation plants, application size and complexity cannot be avoided, and so it is possible for very significant difficulties to occur, when attempting to provide a reliable and fault free control system. Less critical, but of significance nevertheless to many users of processing systems, is the reliability of applications used continuously and widely in the workplace, such as word processing systems, or even operating systems. Furthermore, Internet and telecommunications systems are significantly complex, and increasing amounts of financial and other valuable data are transferred over them. Functional flaws in this environment are, in effect, security flaws which may be exploited.

Summary of the Invention

It is an object of the present invention to provide an improved method of initialising an application comprising a large number of application modules.

According to an aspect of the invention, there is provided apparatus for processing data, comprising processing means and memory means for storing data and instructions for processing said data. The memory means includes application instructions and data that define an initialisation manager and a plurality of application modules. Each of the application modules includes a registration object for registering dependency of said module upon others of said application modules, to said initialisation manager. Each application module further includes operational instructions defining operations of said module used by other modules; and at least two of said application modules include initialisation instructions for initialising data affecting execution of said operational instructions. The initialisation manager includes instructions for performing the steps of: (a) processing said registered module dependencies to identify a dependency count for each module; (b) generating an initialisation schedule by sorting the module order according to the number of dependencies; and (c) calling said initialisation instructions in the order defined by said initialisation schedule.

Brief Description of the Several Views of the Drawings

Figure 1 shows an image processing system, including a computer and a monitor;

Figure 2 summarises actions performed on the processing system shown in *Figure 1*, including a step of starting the application, a step of using the application with plug-ins and a step of closing down the application;

Figure 3 details the components of the processing system shown in *Figure* 1, including a main memory;

Figure 4 summarises application modularity as it relates to the present invention;

Figure 5 summarises the invention, detailing the step of starting an application, shown in *Figure 2*, including a step of loading all application modules, a step of processing dependencies, a step of generating an initialisation schedule and a step of initialising modules;

Figure 6 details the contents of the main memory shown in *Figure 3*, including an initialisation list, an initialisation schedule, and a registration object;

Figure 7 details the step of loading all application modules shown in *Figure 5*;

Figure 8 illustrates C++ source code used to generate the registration object shown in *Figure 6*;

Figure 9 details an example of the initialisation list shown in *Figure 6*;

Figure 10 summarises the step of processing dependencies shown in *Figure 5*, including a step of filling a dependency matrix and a step of processing the dependency matrix;

Figure 11 illustrates the result of the step of filling a dependency matrix shown in *Figure 10*;

Figure 12 details the step of processing the dependency matrix, shown in *Figure 10*;

Figure 13 illustrates the result of processing the dependency matrix as summarised by the steps shown in *Figure 12*;

Figure 14 details the initialisation schedule shown in *Figure 6*;

Figure 15 details the step of initialising modules shown in *Figure 5*;

Figure 16 details the step of operating the application shown in *Figure 2*;

Figure 17 illustrates the effect of loading plug-ins on the initialisation schedule as the application is used;

Figure 18 details the step of closing down the application shown in *Figure 2*;

Figure 19 illustrates C++ source code for a key section of the main executable module of the application; and

Figure 20 illustrates C++ code used to implement aspects of the initialisation process shown in *Figure 5*.

Detailed Description of The Preferred Embodiment

Figure 1

A system for processing image and other data is illustrated in *Figure 1*. A processing system **101**, such as an Octane™ produced by Silicon Graphics Inc., supplies image signals to a video display unit **102**. Moving image data is stored on a redundant array of inexpensive discs (RAID) **103**. The RAID is configured in such a way as to store a large volume of data, and to supply this data at a high bandwidth, when required, to the processing system **101**. The operator controls the processing environment formed by the processing system **101**, the video monitor **102** and the RAID **103**, by means of a keyboard **104**, and a stylus-operated graphics tablet **105**. The processing system shown *in Figure 1* is optimal for the purpose of processing image and other high bandwidth data. In such a system, the instructions for controlling the processing system are complex. The invention relates to any computer system where processing instructions are of significant complexity.

Instructions controlling the processing system **101** may be installed from a physical medium such as a CDROM or DVD disk **106**, or over a network, including the Internet. These instructions enable the processing system **101** to interpret user commands from the keyboard **104** and the graphics tablet **105**, such that image data, and other data, may be viewed, edited and processed.

Figure 2

User operations of the system shown in *Figure 1* are summarised in *Figure* 2. At step **201** the user switches on the computer system. At step **202** application instructions for controlling the processing system **101** are installed if necessary. These instructions may be installed from a CDROM or DVD **106**, or via a network, possibly the Internet. At step **203** the user interacts with the processing system **101** in such a way as to start the application instructions. At step **204** the user interacts with the application now running on the processing system **101**. These interactions include the loading and saving of files.

Files of various formats may be loaded and saved. Each file format has a specific set of instructions for loading and saving. Where a large number of formats is to be loaded and saved, instructions for all formats are not loaded simultaneously. Instead, instructions for format loading and or saving are only loaded when the user initiates an action that explicitly requires them. Instructions of this kind are sometimes referred to as plug-ins, reflecting the fact that a user can obtain such instructions and make them available to the main application according to his or her specific needs.

Plug-ins may provide a broad variety of functionality. In image processing, various types of image filtering, enhancement and modification can be performed by algorithms available as plug-ins. Furthermore, the main application instructions and additional plug-ins need not be written by the same author; they merely need to conform to a standard application programming interface.

At step **205** the user closes the application, and at step **206** the processing system **101** is switched off.

Figure 3

The processing system **101** shown in *Figure 1* is detailed in *Figure 3*. The processing system comprises two central processing units **301** and **302** operating in parallel. Each of these processors is a MIPS R12000 manufactured by MIPS Technologies Incorporated, of Mountain View, California. Each of these processors **301** and **302** has a dedicated secondary cache memory **303** and **304** that facilitate per-CPU storage of frequently used instructions and data. Each CPU **301** and **302** further includes separate primary instruction and data cache memory circuits on the same chip, thereby facilitating a further level of processing improvement. A memory controller **305** provides a common connection between the processors **301** and **302** and a main memory **306**. The main memory **306** comprises two gigabytes of dynamic RAM.

The memory controller **305** further facilitates connectivity between the aforementioned components of the processing system **101** and a high bandwidth non-blocking crossbar switch **307**. The switch makes it possible to provide a direct high capacity connection between any of several attached circuits. These include a graphics card **308**. The graphics card **308** generally receives instructions from the processors **301** and **302** to perform various types of graphical image rendering processes, resulting in images, clips and scenes being rendered in real time on the monitor **102**. A high bandwidth SCSI bridge **309** provides an interface to the RAID **103**, and also, optionally, to a digital tape device, for use as backup.

A second SCSI bridge **310** facilitates connection between the crossbar switch **307** and a DVD/CDROM drive **311**. The DVD drive provides a convenient way of receiving large quantities of instructions and data, and is typically used to install instructions for the processing system **101** onto a hard disk drive **312**. Once installed, instructions located on the hard disk drive **312** may be fetched into main memory **306** and then executed by the processors **301** and **302**. An input output (I/O) bridge **313** provides an interface for the graphics tablet **105** and the keyboard **104**, through which the user is able to provide instructions to the processing system **101**.

Application instructions running on the processing system **101** are complex. Whether the application is a word processor, image editor or a digital film editor, the instructions that define the application's functionality typically run into hundreds of thousands, if not several millions, of individual binary instructions for the processors **301** and **302**. Definition of these instructions is achieved by the use of a high level language such as C++, which is compiled into binary machine code compatible with the intended target processor. However, the use of a high level language, while reducing the effort required to define instructions, still does not solve the complexity problem entirely. As high level languages have become more sophisticated, this has allowed engineers to create more complex applications. The requirement of organisation still imposes a limit upon the complexity that application instructions can achieve. This complexity is minimised by splitting up an application into a large number of modules.

Figure 4

A particular difficulty encountered when combining modules in a single application is that of module initialisation. In *Figure 4*, an illustration is shown of the relationships between modules that lead to this difficulty. An application **401** comprises an executable module **411** and several other modules **412** to

418. The modules **412** to **418** are dynamically loaded shared objects. Under Unix type operating systems, such as Irix™ and Linux™, dynamically shared objects are usually abbreviated as dso's. They are also known simply as shared objects. Under Windows™ operating systems, dso's are known as dynamically loaded libraries, or dll's. The executable module **411** defines the starting point of the application instructions **401**, while the other modules **412** to **418** provide additional functionality that is invoked via the executable **411**.

Each module **411** to **418** includes instructions **421**, in the form of several functions **421**, and data structures **422**. There are two types of data that it is necessary to consider. The first type of data is user data, supplied usually from files on the hard disk **312**, and which is to be created, manipulated and stored by the application. User data includes word processing files, text files, image files and so on. However, from an engineer's perspective a second type of data exists, which has an effect on the behaviour of the application, and the functions **421** within each module. These types of data are indicated at **422**.

An example of this type of data is a mathematical function which is required to calculate a sine function at high speed. A look up table includes a number of pre-calculated results, thus reducing significantly the time required for the function to execute. Thus a data structure affects a function in a module. In some implementations the data structure is created when the application starts, by invoking an initialisation function prior to the start of the main application. In order for the application to function correctly, it is essential for the sine function to be initialised before the application begins. This is an example where data structures **422** within modules must be initialised.

A second requirement for initialisation is when hardware components of the processing system **101** are to be initialised. For example, the graphics card **308** has the potential to operate in a variety of display modes. The application may require a particular mode to operate. In order to do this, a graphics interface module **417** can be used to interrogate the graphics card **308** and determine which of its available modes is suitable for the application. The graphics card is then instructed to operate in the selected mode. Thereafter, many other modules will require information about the selected graphics mode in order to function correctly within the overall context of the application. This is an example of a requirement for hardware initialisation, which also results in data structures being initialised, that represent the behaviour and characteristics of the graphics card so that other modules may function in an appropriate way.

Various types of modules require initialisation, although it is possible that some modules will require none. The situation is made complex by the fact that the order in which modules are initialised is important. For example, if the fast sine function's characteristics are dependent upon the graphics resolution set on the graphics card, it will be necessary for the graphics module to be initialised before the module containing the sine function. In an application comprising ninety or so separate modules, sufficient dependency of this sort exists, that defining the order of initialisation is extremely difficult to achieve. A dependency graph is illustrated at **431**, in which module A is dependent upon modules B and C, module B dependent upon modules D and F and so on. This may be represented as a dependency list, as shown at **441**.

As the application modules undergo daily evolution, through modification and or improvement, these dependencies change also. However, this requires the engineers working on an individual module to be aware of the network of dependencies, and therefore on the overall structure of the application. In the art, the order of initialisation of modules is determined manually by an engineer or engineers, who then write instructions to perform initialisation in the appropriate order. This process is sufficiently complex that trial and error may often be used to determine the most reliable initialisation order. This leads to an reduction in reliability, since it is possible that an incorrect initialisation order will only be exposed by a crash that occurs under very rare conditions.

This is how instruction modularity reaches its complexity limit. Above a certain number of modules, the requirement to identify the initialisation order becomes impossible to meet reliably. The reduction in complexity achieved by splitting the application into modules only works for applications below a certain size, or for applications comprising modules that do not have to be initialised in any particular order. Without these restrictions, application quality and reliability are compromised.

Figure 5

A preferred embodiment of the invention is summarised in *Figure 5*, which highlights the step **203**, of *Figure 2*, in which the application is started. At step **501** an operating system running on the processing system **101** performs loading of all application modules. As a result of this process, each module is registered in a list, along with its dependencies. At step **502** the dependencies are processed. At step **503**, a question is asked as to whether the module dependencies are valid. This condition is known as a result of the processing carried out in step **502**. If the dependencies are not valid, control is directed to step **504**, at which point the application launch is cancelled. Alternatively, control is directed to step **505**.

At step **505** an initialisation schedule is generated by sorting the modules in order of the number of their dependencies. In this step, the number of dependencies is higher than that provided by the list of registered modules at step **501**. This increment is the result of dependency processing performed in step **502**. At step **506** the modules are initialised in the order defined by the initialisation schedule, and at step **507** the post-initialisation application processing begins.

Figure 6

As a result of the processing performed by the steps in *Figure 5*, the contents of main memory **306** are as shown in *Figure 6*. The operating system that performed the loading resides in main memory as indicated at **601**. The application also is resident in main memory as indicated at **602**. Application data **603** includes data loaded by default for the application, possibly including image data, and other data that the application will process, display and or modify. System data **604** includes data used by the operating system **601**. The operating system is Irix™, available from Silicon Graphics Inc.

The application **602** comprises around eighty to ninety application modules, including an executable **605** and an initialisation manager **606**. The initialisation manager **606** and an example application module **607** are shown in detail. The initialisation manager includes an initialisation list **608** and an initialisation schedule **609**. These are generated as a result of the steps shown in *Figure 5*. The application module **607** includes a registration object **610**, an initialisation function **611** and a finalisation function **612**. Operational functions **613** include all the main operations carried out by the module **607** that are not related to initialisation. These functions **611**, **612** and **613** comprise sets of instructions that are executable on the processors **301** and **302**. Initialised data **614** includes data in the module **607** that affects the operation of its functions **613**. Initialised data has to be initialised by the initialisation function **611** before the operational functions **613** can be used. Not all modules necessarily require initialisation, but it is assumed that they do, so as not to restrict the evolution of the application as these types of functions are added freely to modules as required.

Figure 7

The step **501** of loading all application modules, initially shown in *Figure 5*, is summarised in *Figure 7*. When the application first starts, the initialisation manager **606** includes an empty initialisation list. When executed, the steps shown in *Figure 7* result in this list being filled.

The actions of the steps shown in *Figure 7* are performed in accordance with known activities of operating system application loading procedures. However, the existence of the registration object **610** in each module causes these steps, nevertheless, to form a part of the invention. When step **701** is first encountered, the executable module **605** is loaded. At step **702**, the first other module referenced inside the executable module is identified. At step **703** a question is asked as to whether this module is already loaded. If so, control is directed to step **705**. Alternatively, at step **704**, this referenced module is loaded, and the steps of *Figure 7* are executed for that module recursively. At step **705** a question is asked as to whether another module is referenced inside the present module. If so, control is directed to step **702**, and the next referenced module is loaded as necessary. Alternatively, once all referenced modules within the present module have been considered, control is directed to step **706**.

Within each of the application modules there is a registration object **610**. The loading process automatically executes constructors for all the statically declared objects in each module. This occurs at step **706**, as a result of standard operating system procedures. Each registration object's constructor contains instructions to add the module to the initialisation list, along with its dependencies. This occurs before main application processing begins. As a result of the recursive execution of the steps of *Figure 7*, all modules will have registered their existence and their dependencies, and the initialisation list **608** will have been filled. The order of the initialisation list at this stage is not important.

Figure 8

An edited example of the source code for a module containing a registration object is shown in *Figure 8*. The source code is written in C++, but defines the actions of events that occur during the loading of a module which may be implemented in binary processor instructions or any other appropriate instruction medium, and which may be stored on a hard disk, optical disk, or transferred as a stream over a network to facilitate an initialisation process. Furthermore, the example in *Figure 8* is heavily edited to convey the essence of the invention. Proper engineering practice will result in these features being placed in several files, including header files, and a source code file dedicated to initialisation functionality alone, as will be appreciated by those skilled in the art.

In the source code listing, a registration object **610** is declared at **801**. Because this is declared outside any function or other type of structure, it is static. That is to say, it exists from the time the module is loaded to the time the module is unloaded. At **802** a constructor for the template class invoked at **801** is defined. The constructor for a static object is called automatically by the loading process at step **706**. Thus, even before the main application starts, the constructor for each module is called. Any instructions may be placed here, and to implement the invention, a function, addDependency() is called. Its argument, contained in brackets, defines an additional dependency for the present module. The constructor shown in this example has three lines of code, each defining a dependency. Thus, as a result of executing these three lines of code, an entry will have been made in the initialisation list equivalent to:

foo -> database, osal, initialize

meaning that module foo is dependent upon modules database, osal and initialize.

For completeness in this example, code for the performInitialize() **803** and performFinalize() **804** functions is also shown, resulting in compiled functions **611** and **612** shown in *Figure 6*.

Figure 9

The initialisation list **608** is illustrated in *Figure 9*. An extract of the dependencies is shown, not including all eighty modules, and naming each module A, B, C and so on, for convenience. The equivalent dependency graph for this small number of modules is shown at **901**. Even this small fragment of twelve modules illustrates the complexity problem. In reality, the relationships are between in the region of eighty or ninety modules. Potentially even several hundred. A further difficulty may arise in that a cyclic dependency may exist, as shown at **902**. This can be written as:

B -> D -> W -> B

If such a situation exists it needs to be identified, as it is not possible to correctly initialise an application that includes this kind of configuration.

Figure 10

The step **502** of processing dependencies, shown in *Figure 5*, is detailed in *Figure 10*. At step **1001** a square dependency matrix is created. It comprises an array of N by N locations, where N is the number of modules registered in the initialisation list **608**. Each of the locations in the matrix can take the value of TRUE or FALSE. Each row in the matrix defines dependency information for a single module. Thus, the row for module A, has locations set at TRUE or FALSE in each of the columns corresponding to B, C, D and so on. Initially the dependency matrix is all clear. At step **1002** the known dependency information from the initialisation list is used to set the entries in each column accordingly. Additionally, the number of dependencies for each module is recorded.

The dependency information provided by the initialisation list is non-transitive. If module A is dependent upon module B and module B is dependent upon module C, then the fact that module A is ultimately dependent upon module C is not recorded. This type of indirect dependency is known as a transitive dependency. At step **1003** an algorithm known as the Warshall algorithm is used to set locations in the matrix in accordance with transitive dependencies. Cyclic dependencies are also detected. The number of dependencies recorded for each module is increased accordingly. At the end of step **1003**, the dependency matrix includes transitive and non-transitive dependencies, and dependency totals for each module that include both types of dependencies. If, as a result of the steps shown in *Figure 10*, a cyclic dependency is found, this results in question **503** in *Figure 5* being answered in the negative.

Figure 11

The state of the dependency matrix after the completion of step **1002** in *Figure 10*, is illustrated in *Figure 11*. Each row of the dependency matrix **1101** records the non-transitive dependencies defined by the initialisation list **608** shown in *Figure 9*. A set of dependency totals **1102** for each module is also shown. The illustration is an example only. In practice the number of rows and columns is much higher.

Figure 12

The Warshall procedure applied in step **1003** in *Figure 10* is detailed in *Figure 12*. This algorithm is used in graph theory, where the condition of transitive closure provided by the process can provide information about relationships between sets of connected vertices. At step **1201** a variable N is set to equal the number of registered modules. At step **1202** an outer loop is commenced, indexed by a variable Z. At step **1203** a question is asked as to whether the module whose dependencies are represented by row Z has any dependencies at all. If answered in the negative, control is directed to step **1216**. This exclusion of modules from the outer loop can save valuable processing time. At step **1204** a middle loop is initiated, indexed by the variable X. At step **1205** a question is asked as to whether X is equal to Z. If so, there is no need to process this level any further, as it would only consider a situation where the module is dependent upon itself. At step **1206**, it is known that X and Z are different, and a question is asked as to whether the module indexed at row X is dependent upon the module indexed at column Z. This condition is indicated by the matrix at the confluence of this row and column being set to a Boolean value of TRUE. The question may be expressed in the form:

$$X \rightarrow Z\ ?$$

If this is not true, then there is no need to consider module X in this loop any further, and control is directed to step **1215**. If module X is dependent upon module Z, control is directed to step **1207**.

At step **1207**, an inner loop is set up, indexed by a variable Y. At step **1208** a question is asked as to whether Y is equal to Z. If so, this indicates a consideration as to whether a module is dependent upon itself, which is not considered of interest. In this case, control is directed to step **1214** and another value for Y is taken for the inner loop. Alternatively, if Y and Z are different, a question is asked as to whether module Z is dependent upon module Y:

Z -> Y ?

If this condition does not exist, control is directed to step **1214**, and another value for Y is taken. Alternatively, control is directed to step **1210**. At step **1210** two conditions are known, from which a conclusion may be drawn:

X -> Z
and Z -> Y
so X -> Z -> Y
and therefore X -> Y

At step **1210** a transitive dependency has been identified from an analysis of the dependency matrix. In the matrix, the location at row X and column Y is set to a value of TRUE. At step **1211** a question is asked as to whether X is equal to Y, which would indicate a dependency cycle of the form:

X -> Z -> X

If this is the case, a note is made that a dependency cycle has been found, and control is directed to step **1214**. Alternatively, if the dependency is non-cyclic, the number of modules that module X is dependent upon is incremented. Thereafter, the inner, middle and outer loops are continued at steps **1214**, **1215** and **1216** respectively.

Figure 13

Once the algorithm of *Figure 12* has completed, all transitive and non-transitive dependencies will have been recorded by the dependency matrix **1101**, and the total number of dependencies recorded for each module. An example of the type of result that can be expected is shown in *Figure 13*, based upon the preceding example shown in *Figure 11*. Here it will be noted that the number of TRUE entries in the matrix has increased, and the dependency counts also have increased for most of the modules.

Figure 14

Processing at this stage has now reached step **505** *in Figure 5*, or, if a cycle was found, step **504**. With the number of dependencies, direct and indirect, known for all the modules, it is now possible to sort them into dependency order. At step **505**, this sorting process is performed, resulting in the creation of an initialisation schedule **609**, as illustrated *in Figure 14*. At the top of the list is the module with the least dependencies. Initialisation may proceed in the order defined by the schedule, and this is performed at step **506** in *Figure 5*.

Figure 15

The initialisation procedure performed at step **506** is detailed in *Figure 15*. At step **1501** the first module in the initialisation schedule **609** is selected. At step **1502**, the initialisation function **611**, **803**, is called for the selected module. This has the result that the data **614** in the module, upon which correct module functionality depends, is initialised before other modules attempt to use the operational functions **613**. At step **1503** a question is asked as to whether initialisation is complete. If not, control is directed to step **1501**, where the next module in the initialisation schedule **609** is selected. All the application modules are thereby initialised in the order required by their dependency. Dependency characteristics for each module may vary over time, as new features are added and improved, without causing difficulty in identifying the correct initialisation order for the application.

The application modules may be augmented by additional modules at any time while it is running. Modules may be dynamically loaded in response to specific user requirements. It is even possible that a user may download a module from the Internet and use it with the application without stopping and restarting the application. A typical application of this is in import and export filters for different file formats. In image processing, for example, there are many formats in which image data can be stored, including several varieties of compressed image format, such as JPEG. A comprehensive set of filters for all image formats could take up a significant amount of main memory, and also take some considerable time to load and initialise if they were all considered as application modules. A solution to this difficulty is in loading such modules on demand. For example, when a user first requires to export to a JPEG format file, a module including JPEG compression instructions can be loaded. It then remains in memory, as it is likely that the user, having used this facility once, will want to use it again before the application is shut down.

Modules loaded in this way are sometimes referred to as plug-ins, as they can implement new functionality for the application simply by the addition of one or two modules, rather than a complete re-installation of the application. Additional use of plug-ins includes image processing algorithms, lens effects, image blur, colour correction and so on. The use of modules, both as standard application modules and in the form of plug-ins that are loaded on demand, is a powerful method of enhancing application functionality, and tailoring it to individual user requirements.

The initialisation framework that has been described can be extended to include all modules loaded at any time during the application's execution. In *Figure 2*, at step **204**, the user performs various actions, including actions that require the loading of plug-in modules.

Figure 16

The steps that occur when plug-ins are loaded are detailed in *Figure 16*. At step **1601** the user performs an action requiring the use of a plug-in module. For example, the user decides to save a file in the JPEG format. At step **1602** A temporary initialisation manager is instantiated, having an empty initialisation list and schedule. At step **1603** the plug-in module or modules are recursively loaded in accordance with the process described in detail with reference to *Figure 7*. A new initialisation list is created, containing only those modules that have not been initialised, even when the plug-in module has a dependency upon modules that were loaded as part of the application at step **203**.

At step **1604** new dependencies are processed as previously described with reference to *Figures 10 to 13*. At step **1605** a question is asked as to whether the module dependencies are valid. If so, control is directed to step **1607**. If a dependency cycle has been found, control is directed to step **1606**, and the plug-in cannot be loaded. An error message is supplied to the user, and, preferably, this information is also supplied back to the vendor of the plug-in for debugging. At step **1607** an initialisation schedule is created, by sorting the modules in order of least dependent, as shown in *Figure 14*. At step **1608** the plug-in modules are initialised, as detailed in *Figure 15*.

At step **1609** the initialisation schedule generated by the newly instantiated initialisation manager is appended to the end of the main initialisation schedule **609**. At step **1610** plug-in processing can begin.

Figure 17

The effect of loading plug-ins on the initialisation schedule **609** is illustrated in *Figure 17*. Each time a new set of modules is loaded, the temporary initialisation schedule created as a result of this process is added to the end of the existing initialisation schedule. This has no effect upon the initialisation of plug-in modules, as their schedules are only added to the main one after their initialisation has been completed. The purpose of recording initialisations in this way is to ensure that finalisation can be performed in a similarly rigorous way.

Finalisation of modules can be as important as initialisation. When an application closes down, various clean-up operations take place. If a function starts behaving incorrectly, because a module that it relied upon has not been closed down correctly, then it is possible for this to cause serious problems. When an application is closed, usually some data is stored to disk. In many cases, incorrect finalisation can cause an application to crash, thus losing this data, and at the very least causing some consternation for the user.

Figure 18

The correct order of module finalisation is the reverse of module initialisation. This is the reason for appending all subsequent plug-in initialisations to the main initialisation schedule. Finalisation, performed in step **205** in *Figure 2*, is detailed in *Figure 18.* At step **1801** the last module in the initialisation schedule **609** is selected. At step **1802** the finalisation function **612**, **804**, is called in the selected module. At step **1803** a question is asked as to whether finalisation is complete. If not, control is directed to step **1801**, and the next last module is selected. This sequence repeats until the first module, at the top of the schedule is reached. Thereafter no more finalisation is required and the application closes.

The loading of modules performed at step **501**, when the application is first loaded, builds the initialisation list **608** automatically as a result of executing the constructor **802** for an registration object **610**, **801**, in each application module. Steps **502** to **507** are not automatic, and must be invoked as the first operation performed by the application when the operating system has completed the loading process.

Figure 19

An application written in C++ usually has its starting point defined in a main() function. This may typically be found in a file called main.cpp. Key features of the source code in this file are shown in *Figure 19.* The file main.cpp defines a module called main, and this therefore has a registration object and constructor as shown in *Figure 8* for other modules. The feature of relevance in *Figure 19* is shown at **1901**. An initialisation object ig is instantiated between pairs of curly braces at **1902** and **1903**. At **1901**, the ig object is created, calling instructions for its constructor. The constructor contains a call to a function that performs the initialisation steps **502** to **506** shown in *Figure 5*. At **1903** the ig object moves out of scope, and it's destructor function is called, thereby invoking instructions for finalisation. Between **1901** and **1903**, the main() function contains the main application functionality.

Figure 20

The constructor for the Initguard class, invoked by the declaration at **1901** in *Figure 19*, is detailed in *Figure 20*. Here it can be seen that a function, located in the initialisation manager module **606**, is called. This function performs the functionality of steps **502** to **506** shown in *Figure 5*.

Figure 21

An example of code used when loading plug-in modules is shown in *Figure 21*. A pair of curly braces **2101** and **2102** defines the scope of a re-initialisation object rig. The constructor for this object increments a reference counter and a temporary empty initialisation list and schedule are created as a result of this declaration. It is possible for re-initialisation functionality to be nested, although this is not usually the case. This corresponds to step **1602** in *Figure 16*. Two dlopen() functions load the plug-ins "Plugin1" and "Plugin2". Loading of these modules results in the temporary initialisation list of rig being filled, in accordance with step **1603** in *Figure 16*. As rig moves out of scope at **2102**, its destructor is called. This is shown in *Figure 22*. This calls a function, ReInitManager.inititialize(), within the initialisation manager module **606**, which has the effect of performing steps **1604** to **1609** shown in *Figure 16*.

The steps summarised by *Figure 5* and detailed thereafter, and also in *Figure 16*, describe events that occur within an initialisation framework. Finalisation is considered part of this framework. The framework exists by virtue of certain data structures being present in the application modules, and any plug-ins that the application requires to use. During the course of application module development, the initialisation manager **606** remains fixed in its operation. Other modules, of course, may vary, as source code in C++ or other language becomes debugged, modified or improved. The engineers responsible for the development of a particular module must write the initialisation and finalisation functions **803** and **804** themselves, as these are entirely dependent upon the intended functionality of the module. However, the declaration **801** of the initialisation object and the contents of its constructor **802** may be generated automatically in response to a list of dependencies supplied by an engineer. A text processing utility, such as sed, may be used to automatically generate detailed code, thus leaving the engineers to concentrate on more creative aspects of the module's design.

Claims

1. Apparatus for processing data, comprising processing means and memory means for storing data and instructions for processing said data, wherein said memory means includes application instructions and data that define

an initialisation manager; and

a plurality of application modules;

each of said application modules includes a registration object for registering dependency of said module upon others of said application modules, to said initialisation manager;

each said application module further includes operational instructions defining operations of said module used by other modules; and

at least two of said application modules include initialisation instructions for initialising data affecting execution of said operational instructions;

said initialisation manager includes instructions for performing the steps of:

(a) processing said registered module dependencies to identify a dependency count for each module;

(b) generating an initialisation schedule by sorting the module order according to the number of dependencies; and

(c) calling said initialisation instructions in the order defined by said initialisation schedule.

2. Apparatus according to claim 1, wherein said registration object

for a module includes registration instructions that are called automatically as a result of loading the module.

3. Apparatus according to claim **1**, wherein said initialisation manager includes an initialisation list that records module dependencies.

4. Apparatus according to claim **1**, wherein said processing step (a) comprises steps of

(a1) creating a dependency array that defines non-transitive dependencies;

(a2) processing said dependency array to identify transitive dependencies; and

(a3) recording the total number of dependencies for each registered module.

5. Apparatus according to claim **1**, wherein said initialisation manager further includes instructions for initialising plug-in modules loaded after step (c), including the steps of:

(d) processing registered plug-in module dependencies to identify a dependency count for each plug-in module;

(e) generating an additional initialisation schedule by sorting the newly registered plug-in modules into order of number of dependencies;

(f) calling initialisation instructions in said plug-in modules in an order defined by said additional initialisation schedule; and

(g) extending the existing initialisation schedule by adding said additional initialisation schedule.

6. Apparatus according to claim **1**, wherein at least two of said application modules include finalisation instructions for finalising data affecting operational instructions.

7. Apparatus according to claim **5** or claim **6**, wherein said initialisation manager includes instructions for finalising modules by calling module finalisation instructions in the reverse of the order defined by the initialisation schedule.

8. Apparatus according to claim **1**, wherein one of said application modules includes the main application instructions, from which a call is made to invoke processing steps (a), (b) and (c), performed by the initialisation manager.

9. Apparatus according to claim **1**, wherein said memory means also includes automatic code generating instructions, for generating source code for an initialisation object in a module.

10. Apparatus according to claim **1**, wherein said application is an operating system.

11. A method of processing data in a processing system comprising processing means and memory means for storing data and instructions for processing said data, wherein said memory means includes application instructions and data that define

an initialisation manager; and

a plurality of application modules;

each of said application modules including a registration object for registering dependency of said module upon others of said application modules, to said initialisation manager;

each said application module further includes operational instructions defining operations of said module used by other modules; and

at least two of said application modules include initialisation instructions for initialising data affecting execution of said operational instructions;

said initialisation manager performing the steps of:

(a) processing said registered module dependencies to identify a dependency count for each module;

(b) generating an initialisation schedule by sorting the module order according to the number of dependencies; and

(c) calling said initialisation instructions in the order defined by said initialisation schedule.

12. A method according to claim **11**, wherein said registration object for a module includes registration instructions that are called automatically as a result of loading the module.

13. A method according to claim **11**, wherein said initialisation manager records said module dependencies in an initialisation list.

14. A method according to claim **11**, wherein said processing step (a) comprises steps of (a1) creating a dependency array that defines non-transitive dependencies;

(a2) processing said dependency array to identify transitive dependencies; and

(a3) recording the total number of dependencies for each registered module.

15. A method according to claim **11**, wherein said initialisation manager further performs steps for initialising plug-in modules loaded after step (c), which include the steps of:

(d) processing registered plug-in module dependencies to identify a dependency count for each plug-in module;

(e) generating an additional initialisation schedule by sorting the newly registered plug-in modules into order of number of dependencies;

(f) calling initialisation instructions in said plug-in modules in an order defined by said additional initialisation schedule; and

(g) extending the existing initialisation schedule by adding said additional initialisation schedule.

16. A method according to claim **11**, including executing finalising instructions contained in at least two of said application modules.

17. A method according to claim **16**~~,~~ wherein said initialisation manager calls said module finalising instructions in the reverse of the order defined by the initialisation schedule.

18. A method according to claim **11**, including making a call from the main application function to invoke processing steps (a), (b) and (c), prior to main application execution.

19. A method according to claim **11**, including executing automatic code generating instructions, thereby generating source code for an initialisation object in response to specified module dependencies.

20. A method according to claim **11**, wherein said application is an operating system.

21. A data structure defined upon a machine readable medium, comprising an initialisation manager and a plurality of application modules; wherein

each of said application modules includes a registration object for registering dependency of said module upon others of said application modules, to said initialisation manager;

each said application module further includes operational instructions defining operations of said module used by other modules; and

a plurality of said application modules include initialisation instructions for initialising data affecting execution of said operational instructions;

said initialisation manager includes instructions for performing the steps of:

(a) processing said registered module dependencies to identify a dependency count for each module;

(b) generating an initialisation schedule by sorting the module order according to the number of dependencies; and

(c) calling said initialisation instructions in the order defined by said initialisation schedule.

22. A data structure according to claim **21**, wherein said registration object for a module includes registration instructions that are called automatically as a result of loading the module.

23. A data structure according to claim **21**, wherein said initialisation manager includes an initialisation list that records module dependencies.

24. A data structure according to claim **21**, wherein said processing step (a) comprises steps of

(a1) creating a dependency array that defines non-transitive dependencies;

(a2) processing said dependency array to identify transitive dependencies; and

(a3) recording the total number of dependencies for each registered module.

25. A data structure according to claim **21**, wherein said initialisation manager further includes instructions for initialising plug-in modules loaded after step (c), including the steps of:

(d) processing registered plugin module dependencies to identify a dependency count for each plugin module;

(e) generating an additional initialisation schedule by sorting the newly registered plugin modules into order of number of dependencies;

(f) calling initialisation instructions in said plugin modules in an order defined by said additional initialisation schedule; and

(g) extending the existing initialisation schedule by adding said additional initialisation schedule.

26. A data structure according to claim **21**, wherein at least two of said application modules include finalisation instructions for finalising data affecting operational instructions.

27. A data structure according to claim **25** or claim **26**, wherein said initialisation manager includes instructions for finalising modules by calling module finalisation instructions in the reverse of the order defined by the initialisation schedule.

28. A data structure according to claim **21**, wherein one of said application modules includes the main application instructions, from which a call is made to invoke processing steps (a), (b) and (c), performed by the initialisation manager.

29. A data structure according to claim **21**, wherein said memory means also includes automatic code generating instructions, for generating source code for an initialisation object in a module.

30. A data structure according to claim **21**, wherein said registered application modules are part of an operating system.

Abstract

A method of initialising application instructions on a processing system. An application (**602**) comprises a number of dynamically shared objects or modules. Each of these modules may include data structures (**614**) that require initialisation. Modules are dependent upon each other, and a module initialisation order is identified by automatically registering a module's dependencies in an initialisation list (**608**) during module loading, processing module dependencies to identify all dependencies, and generating an initialisation schedule (**609**). Module initialisation (**506**) is then performed. Plug-in modules can be loaded and initialised after the application has started, and the plug-in schedule is appended to the initialisation schedule. Finalisation is performed in reverse order, when the application is closed.

(Figure 5)

2034-P553-GB

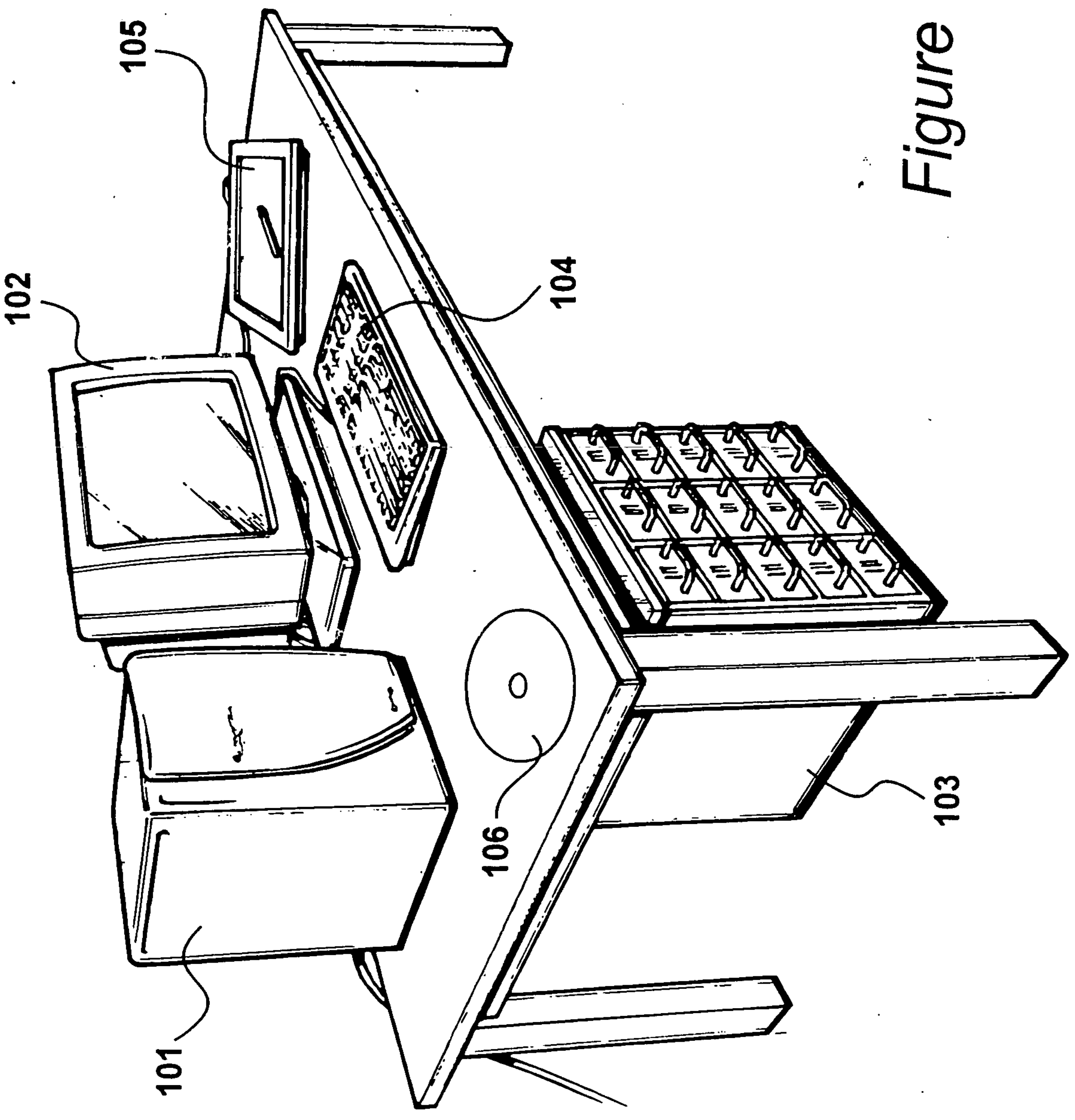


Figure 1

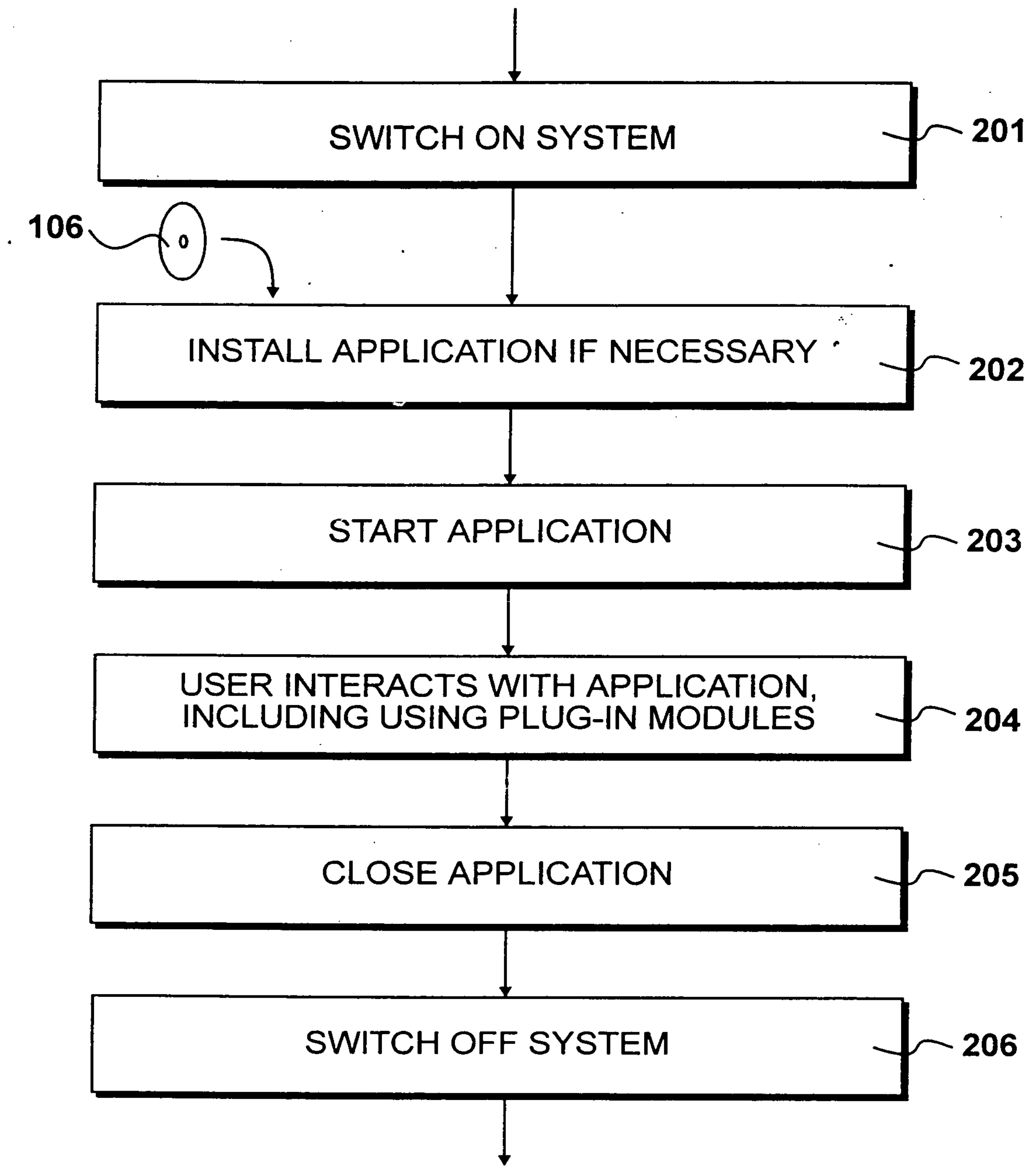

SWITCH ON SYSTEM
201
106
INSTALL APPLICATION IF NECESSARY
202
START APPLICATION
203
USER INTERACTS WITH APPLICATION, INCLUDING USING PLUG-IN MODULES
204
CLOSE APPLICATION
205
SWITCH OFF SYSTEM
206

Figure 2

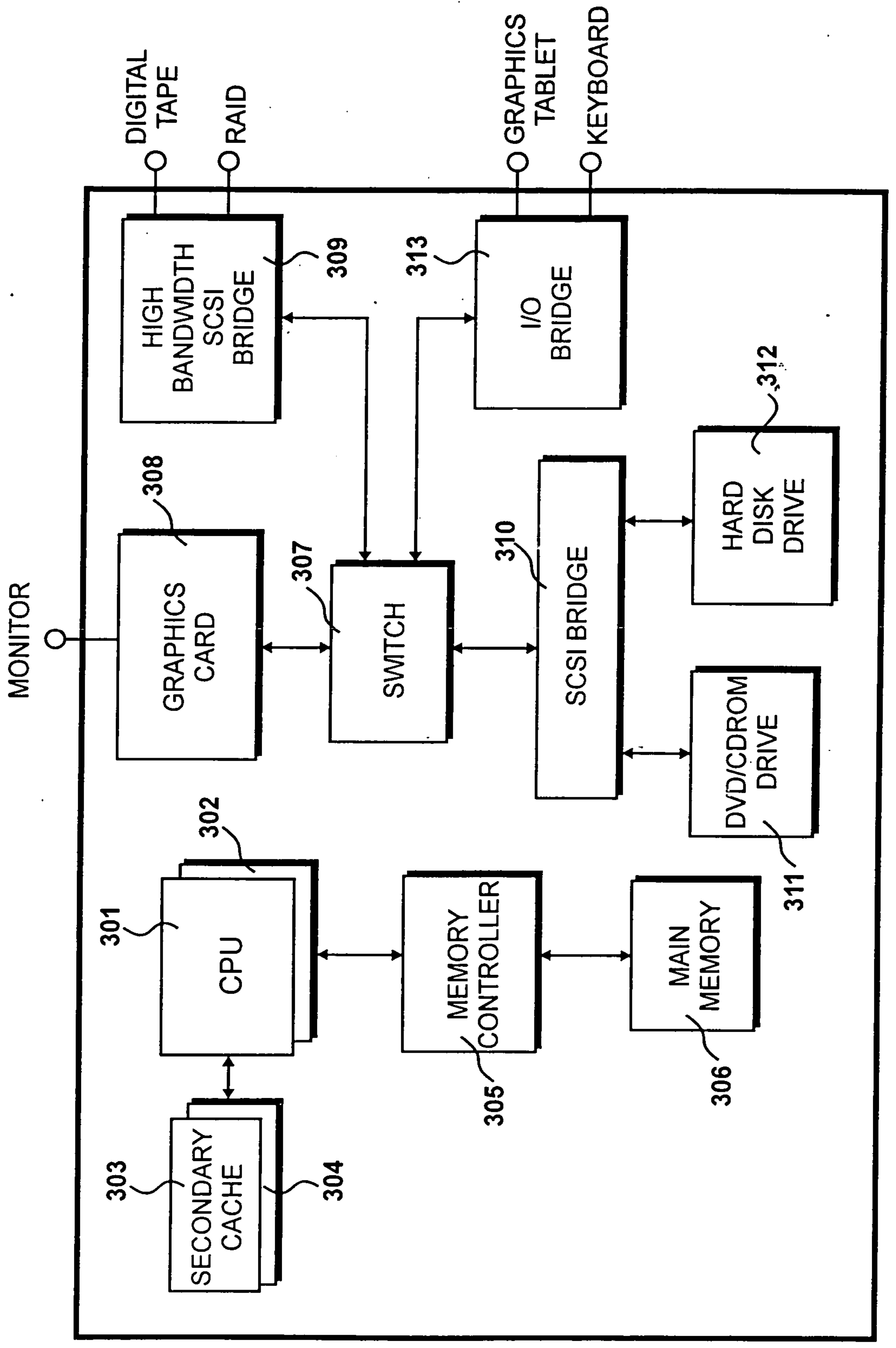
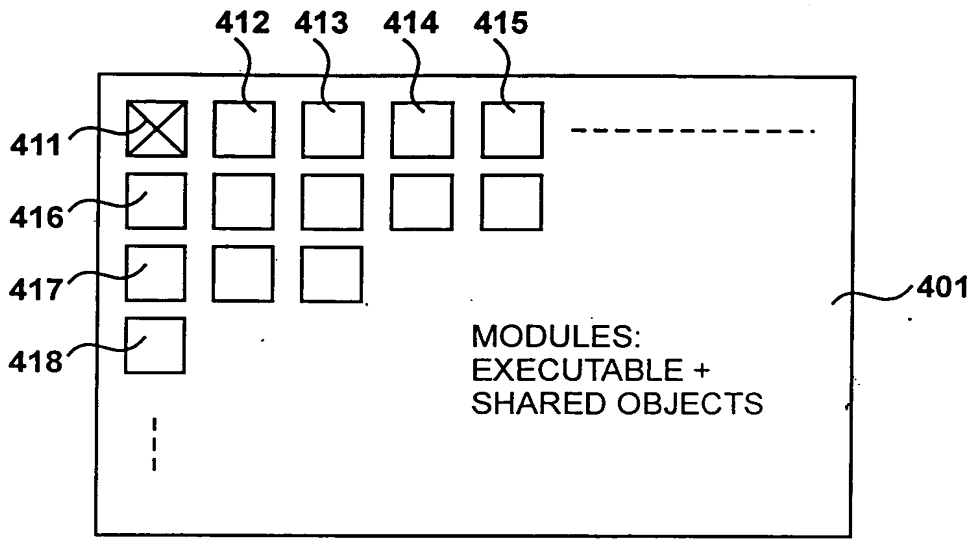
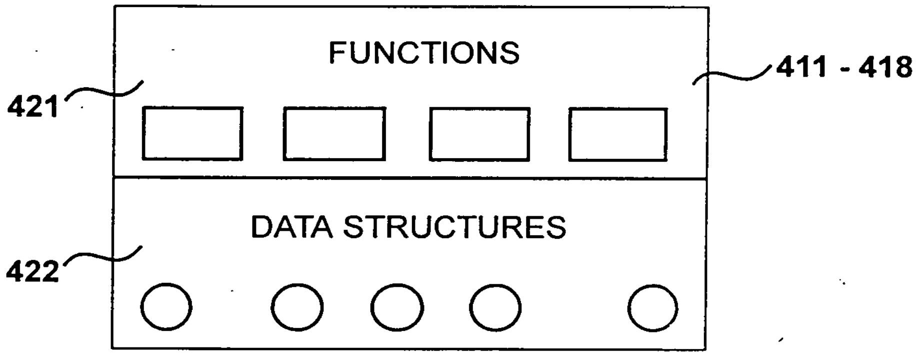

DIGITAL TAPE
RAID
GRAPHICS TABLET
KEYBOARD
HIGH BANDWIDTH SCSI BRIDGE
309
313
I/O BRIDGE
312
HARD DISK DRIVE
MONITOR
GRAPHICS CARD
308
307
SWITCH
310
SCSI BRIDGE
DVD/CDROM DRIVE
311
302
301
CPU
MEMORY CONTROLLER
MAIN MEMORY
305
306
303
SECONDARY CACHE
304

Figure 3


412
413
414
415
411
416
417
418
401
MODULES:
EXECUTABLE +
SHARED OBJECTS


FUNCTIONS
421
411 - 418
DATA STRUCTURES
422

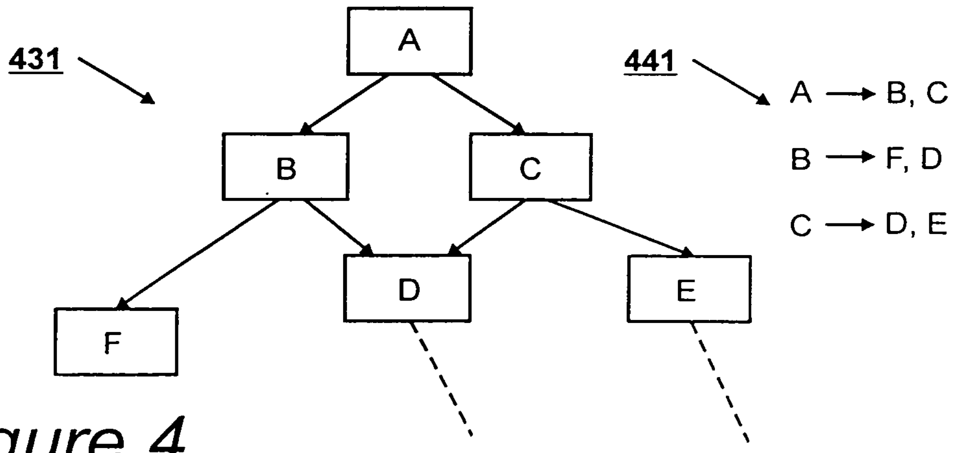
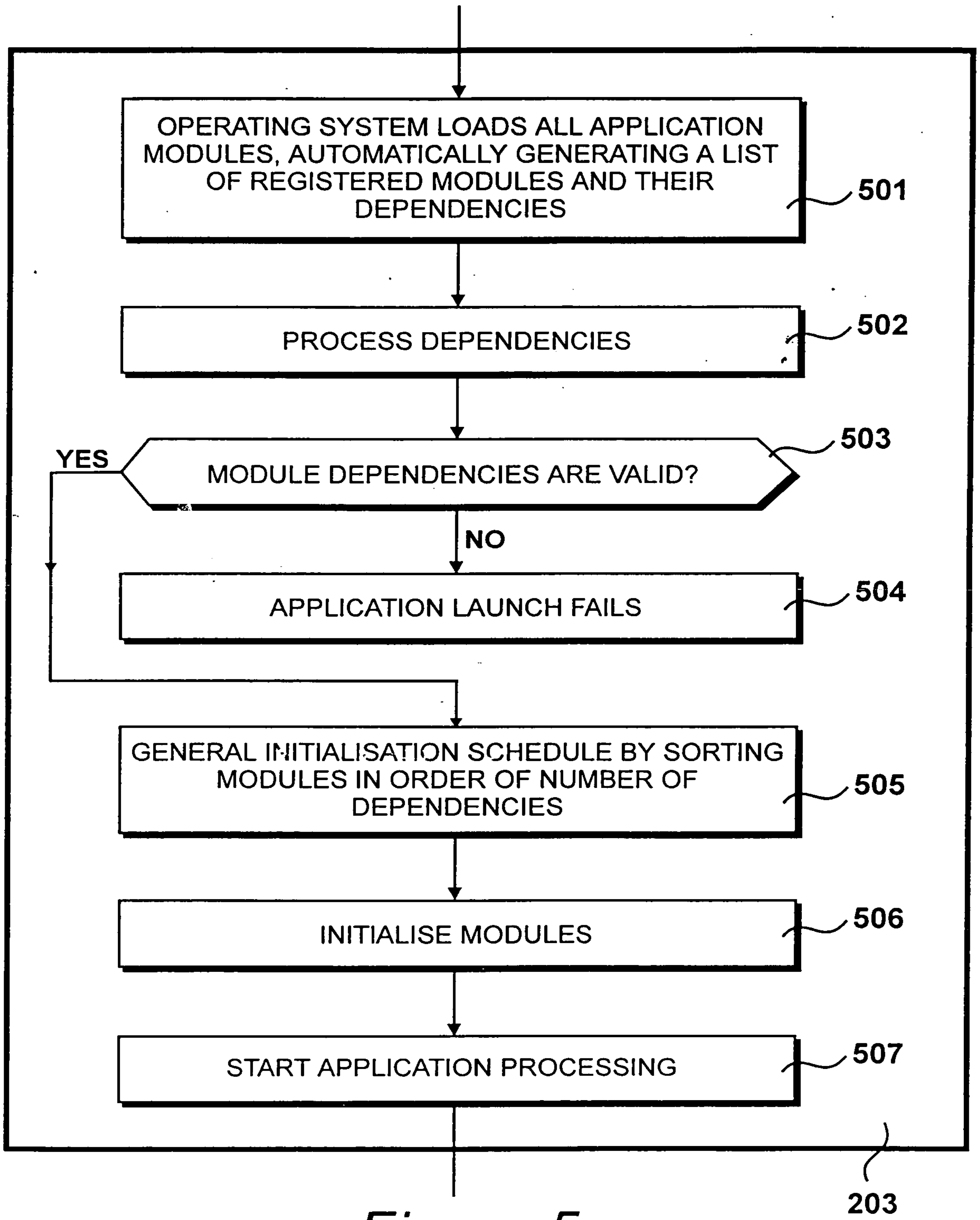

431
A
B
C
D
E
F
441
A → B, C
B → F, D
C → D, E

Figure 4



OPERATING SYSTEM LOADS ALL APPLICATION MODULES, AUTOMATICALLY GENERATING A LIST OF REGISTERED MODULES AND THEIR DEPENDENCIES
501
PROCESS DEPENDENCIES
502
MODULE DEPENDENCIES ARE VALID?
503
YES
NO
APPLICATION LAUNCH FAILS
504
GENERAL INITIALISATION SCHEDULE BY SORTING MODULES IN ORDER OF NUMBER OF DEPENDENCIES
505
INITIALISE MODULES
506
START APPLICATION PROCESSING
507
203

Figure 5

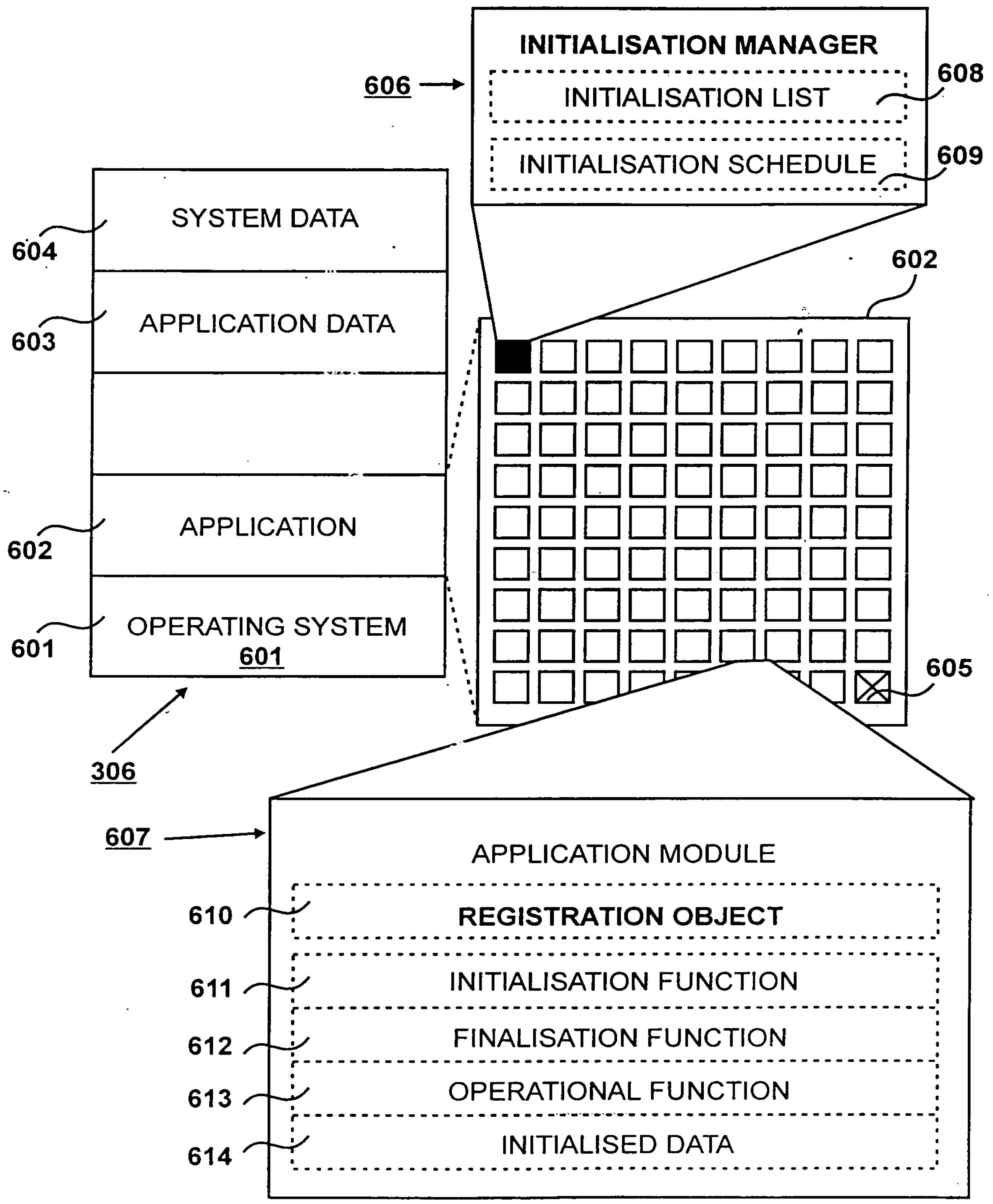

INITIALISATION MANAGER
INITIALISATION LIST
INITIALISATION SCHEDULE
606
608
609
SYSTEM DATA
APPLICATION DATA
APPLICATION
OPERATING SYSTEM
601
604
603
602
601
602
605
306
607
APPLICATION MODULE
REGISTRATION OBJECT
INITIALISATION FUNCTION
FINALISATION FUNCTION
OPERATIONAL FUNCTION
INITIALISED DATA
610
611
612
613
614

Figure 6

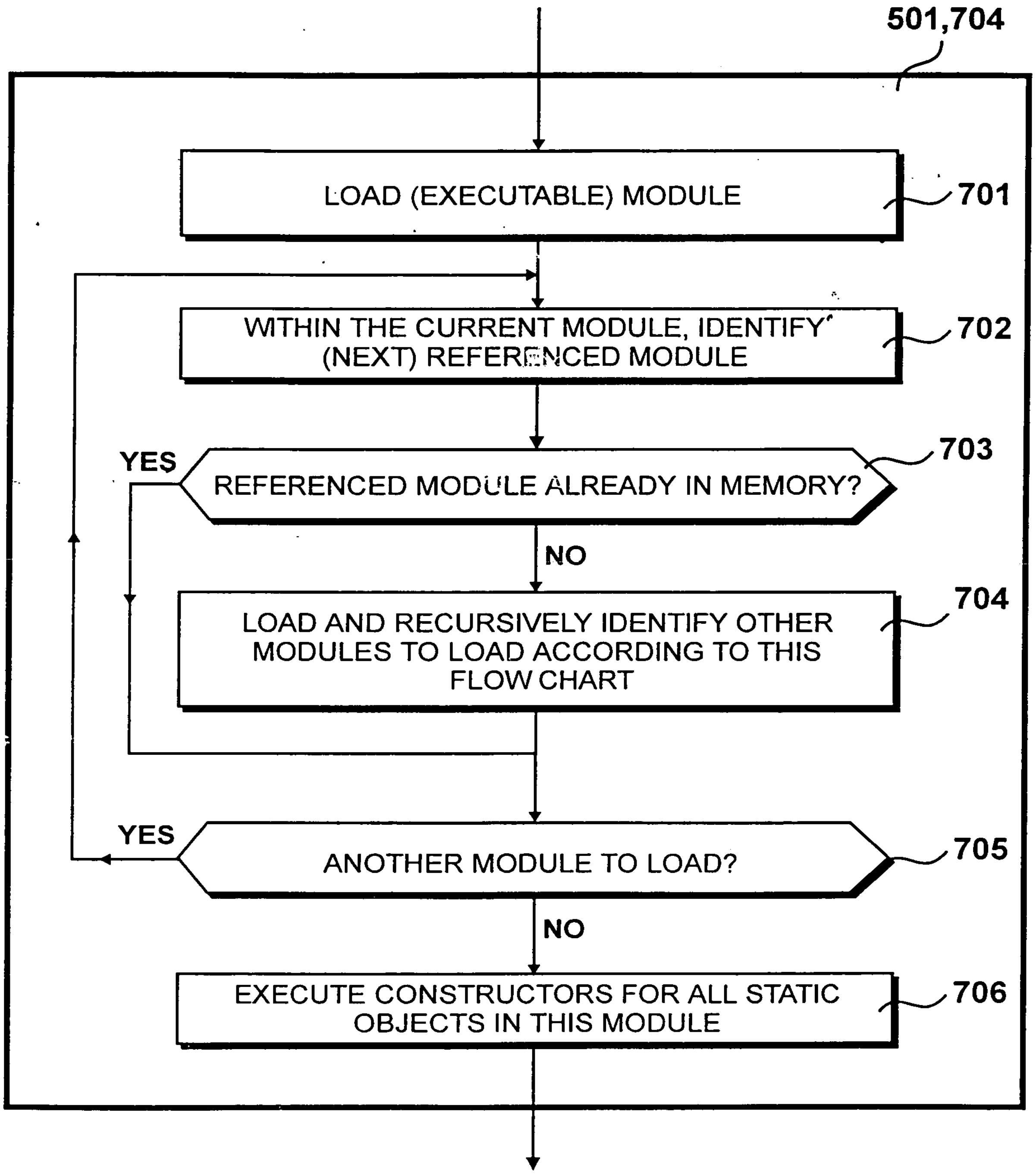
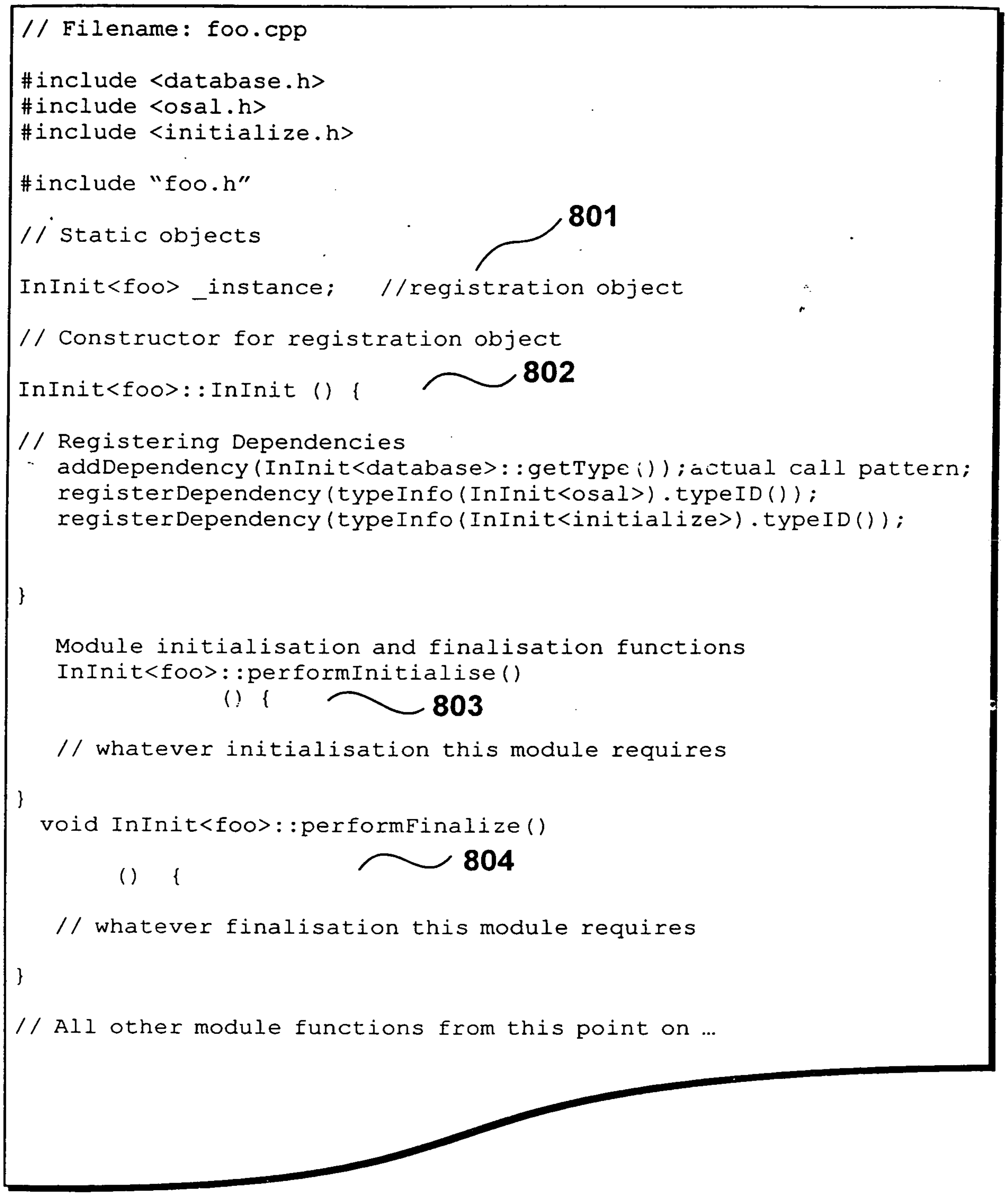

501,704
LOAD (EXECUTABLE) MODULE
701
WITHIN THE CURRENT MODULE, IDENTIFY (NEXT) REFERENCED MODULE
702
REFERENCED MODULE ALREADY IN MEMORY?
703
YES
NO
LOAD AND RECURSIVELY IDENTIFY OTHER MODULES TO LOAD ACCORDING TO THIS FLOW CHART
704
ANOTHER MODULE TO LOAD?
705
YES
NO
EXECUTE CONSTRUCTORS FOR ALL STATIC OBJECTS IN THIS MODULE
706

Figure 7



```
// Filename: foo.cpp

#include <database.h>
#include <osal.h>
#include <initialize.h>

#include "foo.h"

// Static objects

InInit<foo> _instance;   //registration object

// Constructor for registration object

InInit<foo>::InInit () {

// Registering Dependencies
   addDependency(InInit<database>::getType());actual call pattern;
   registerDependency(typeInfo(InInit<osal>).typeID());
   registerDependency(typeInfo(InInit<initialize>).typeID());


}

   Module initialisation and finalisation functions
   InInit<foo>::performInitialise()
               () {

   // whatever initialisation this module requires

}
  void InInit<foo>::performFinalize()

       ()   {

   // whatever finalisation this module requires

}

// All other module functions from this point on …
```

Figure 8

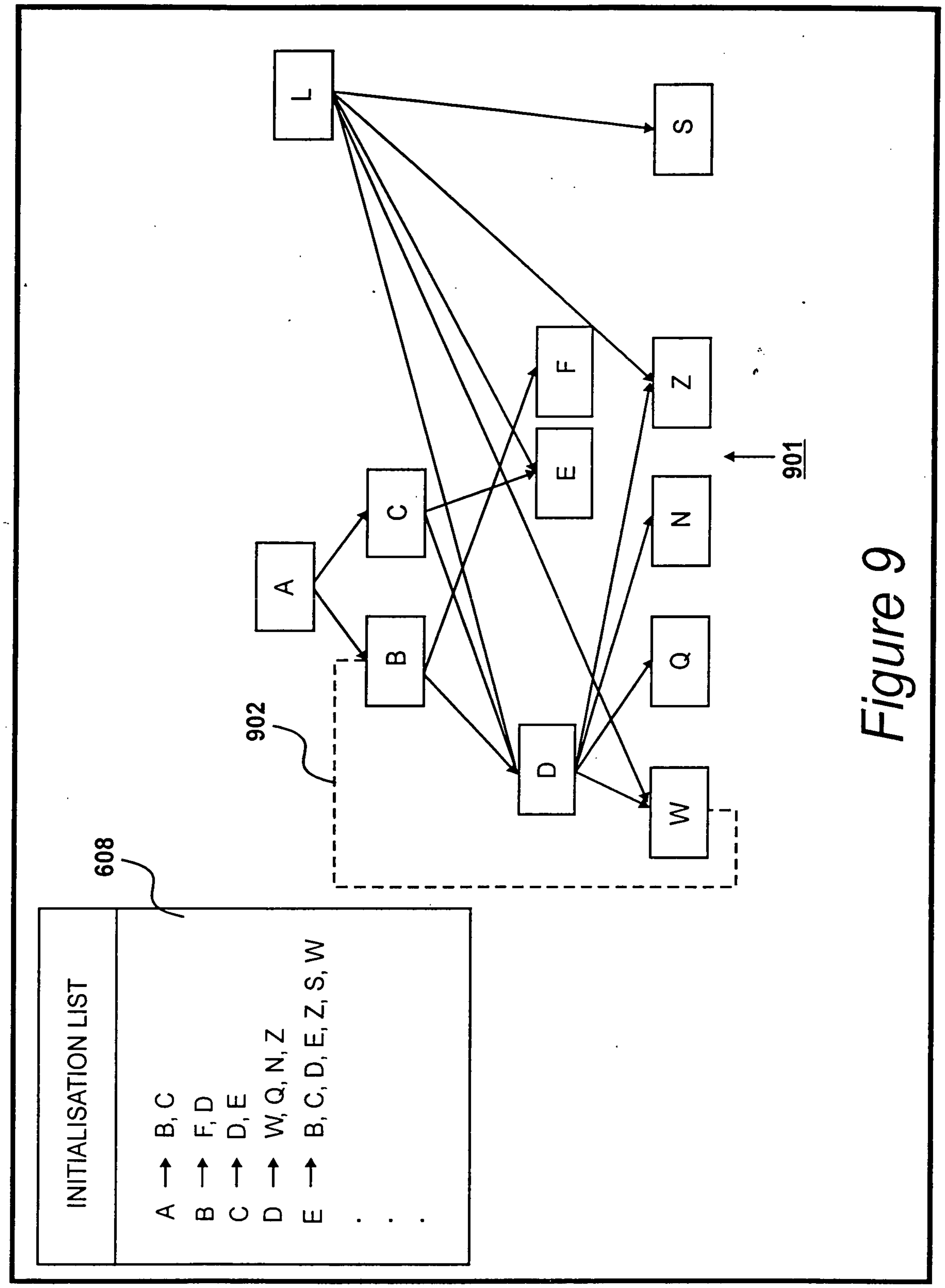

A
B
C
D
E
F
L
S
Z
N
Q
W
901
902
608
INITIALISATION LIST
A → B, C
B → F, D
C → D, E
D → W, Q, N, Z
E → B, C, D, E, Z, S, W
. . .

Figure 9

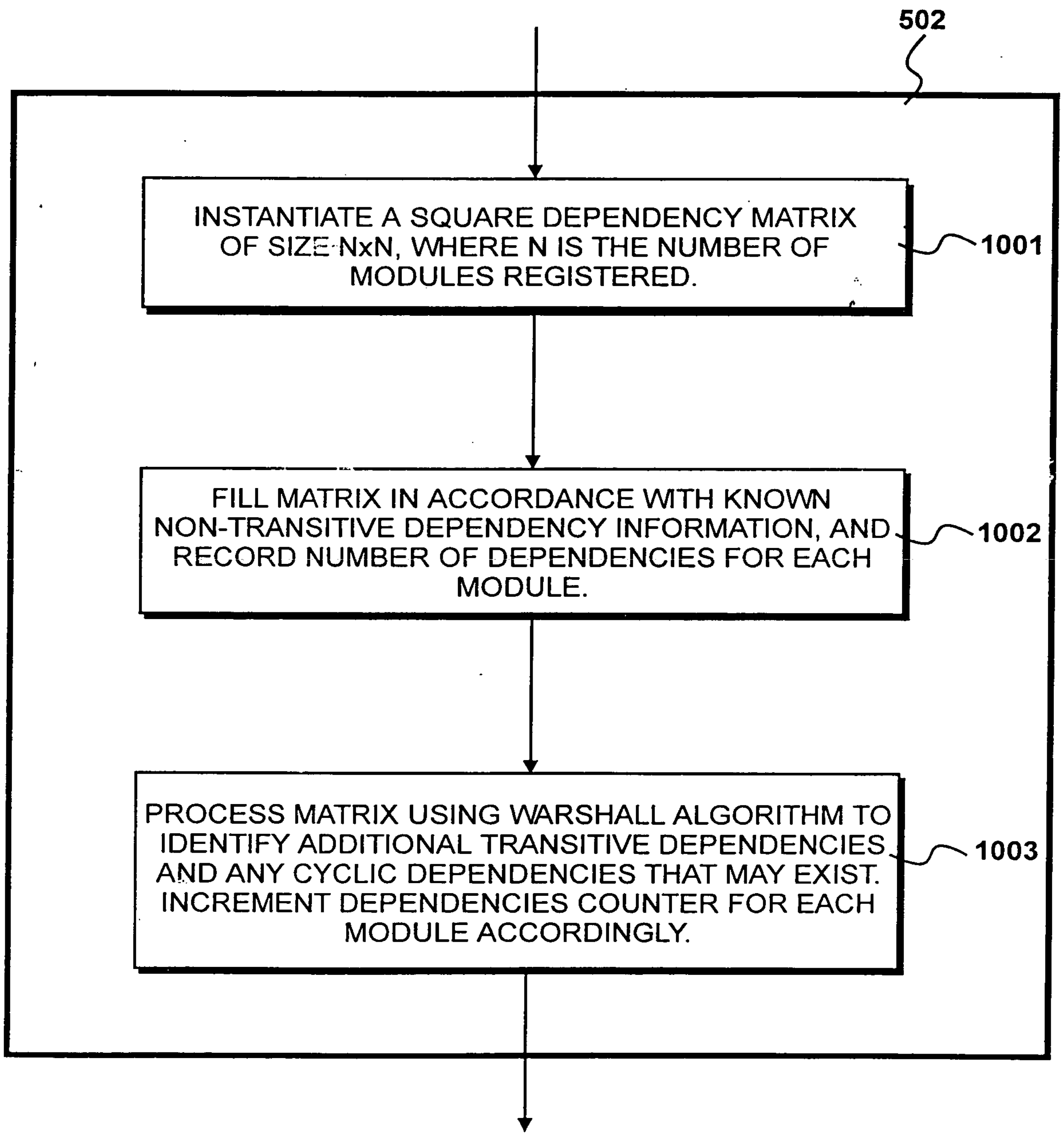
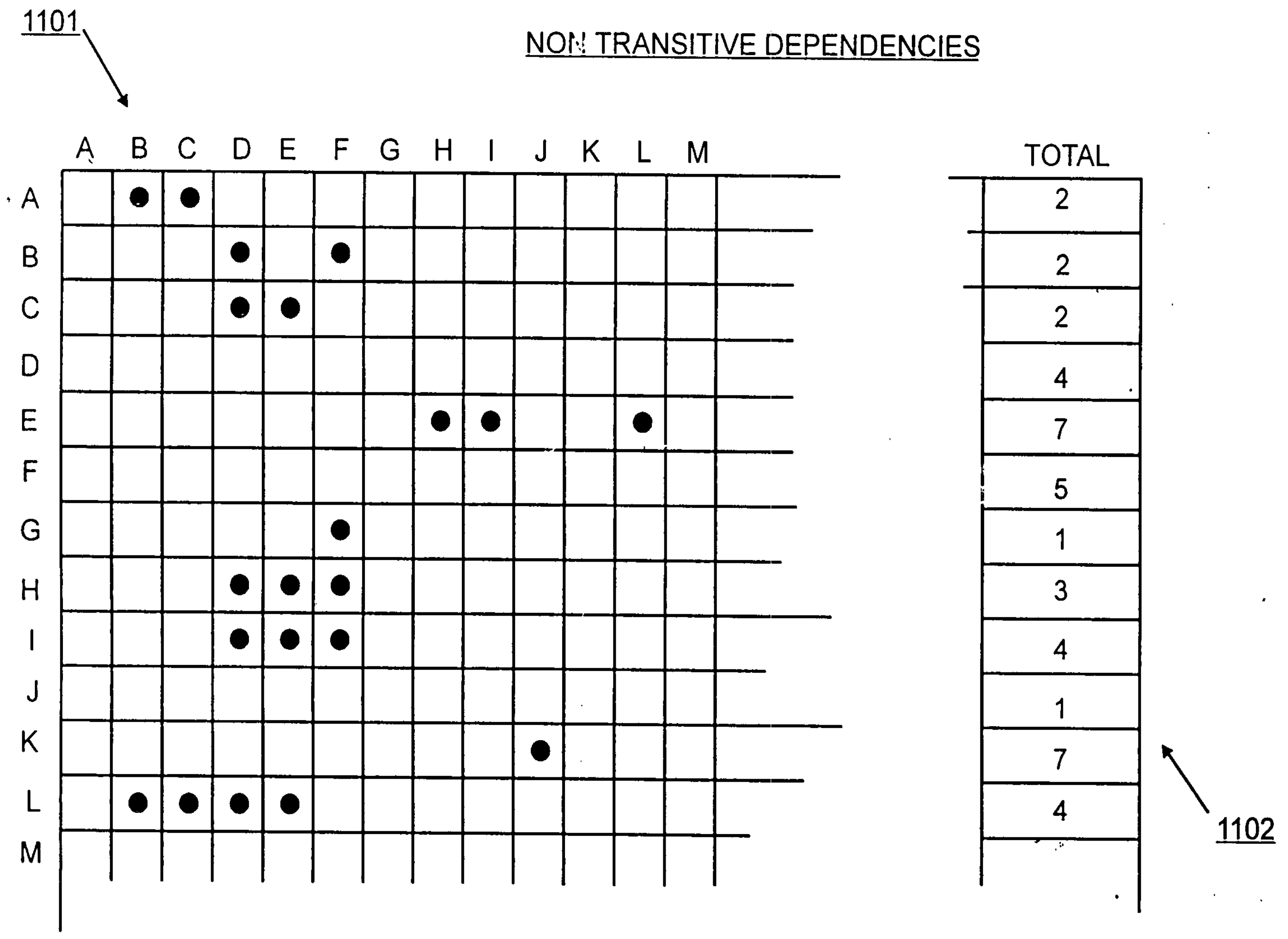

502
INSTANTIATE A SQUARE DEPENDENCY MATRIX OF SIZE NxN, WHERE N IS THE NUMBER OF MODULES REGISTERED.
1001
FILL MATRIX IN ACCORDANCE WITH KNOWN NON-TRANSITIVE DEPENDENCY INFORMATION, AND RECORD NUMBER OF DEPENDENCIES FOR EACH MODULE.
1002
PROCESS MATRIX USING WARSHALL ALGORITHM TO IDENTIFY ADDITIONAL TRANSITIVE DEPENDENCIES AND ANY CYCLIC DEPENDENCIES THAT MAY EXIST. INCREMENT DEPENDENCIES COUNTER FOR EACH MODULE ACCORDINGLY.
1003

Figure 10

NON TRANSITIVE DEPENDENCIES



	A	B	C	D	E	F	G	H	I	J	K	L	M	TOTAL
A		●	●											2
B				●		●								2
C				●	●									2
D														4
E								●	●			●		7
F														5
G						●								1
H				●	●	●								3
I				●	●	●								4
J														1
K										●				7
L		●	●	●	●									4
M														

Figure 11

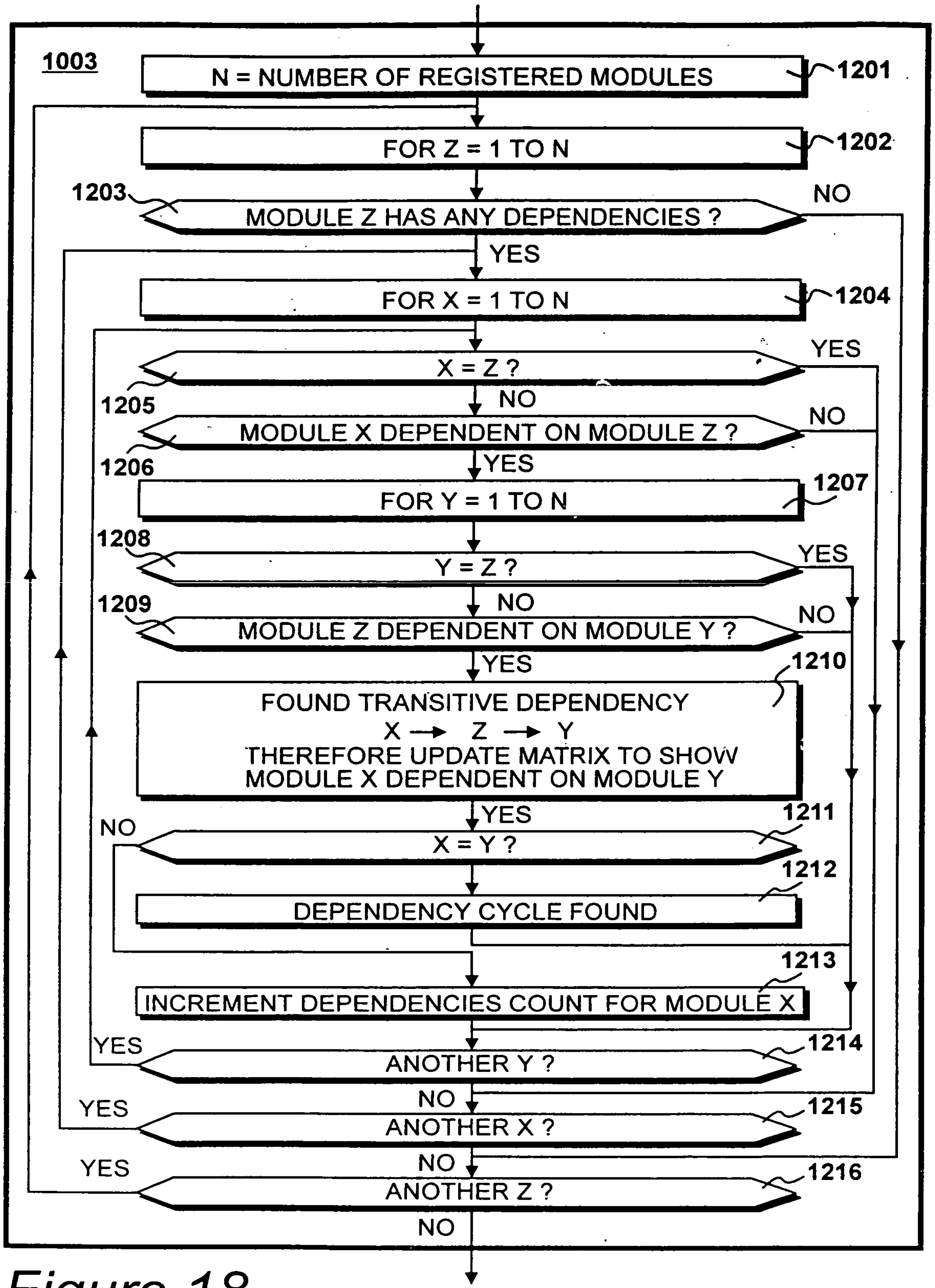

1003
N = NUMBER OF REGISTERED MODULES
1201
FOR Z = 1 TO N
1202
1203
MODULE Z HAS ANY DEPENDENCIES ?
NO
YES
FOR X = 1 TO N
1204
X = Z ?
YES
1205
NO
MODULE X DEPENDENT ON MODULE Z ?
NO
1206
YES
FOR Y = 1 TO N
1207
1208
Y = Z ?
YES
1209
NO
MODULE Z DEPENDENT ON MODULE Y ?
NO
YES
1210
FOUND TRANSITIVE DEPENDENCY
X → Z → Y
THEREFORE UPDATE MATRIX TO SHOW
MODULE X DEPENDENT ON MODULE Y
YES
1211
NO
X = Y ?
1212
DEPENDENCY CYCLE FOUND
1213
INCREMENT DEPENDENCIES COUNT FOR MODULE X
YES
ANOTHER Y ?
1214
NO
YES
ANOTHER X ?
1215
NO
YES
ANOTHER Z ?
1216
NO

Figure 18

1101

NON TRANSITIVE DEPENDENCIES

	A	B	C	D	E	F	G	H	I	J	K	L	M
A		●	●	●	●	●		●	●			●	
B				●		●							
C				●	●			●	●			●	
D													
E								●	●			●	
F													
G						●							
H				●	●	●						●	
I				●	●	●						●	
J													
K										●			
L		●	●	●	●	●		●	●				
M													

TOTAL
23
5
9
15
17
8
1
14
14
11
9
21

1102

Figure 13

INITIALISATION SCHEDULE

MODULE	DEPENDENCIES	RANK
G	1	1
B	5	2
F	8	3
C	9	4
K	9	5
J	11	6
H	14	7
I	14	8
D	15	9
E	17	10
L	21	11
A	23	12

609

Figure 14

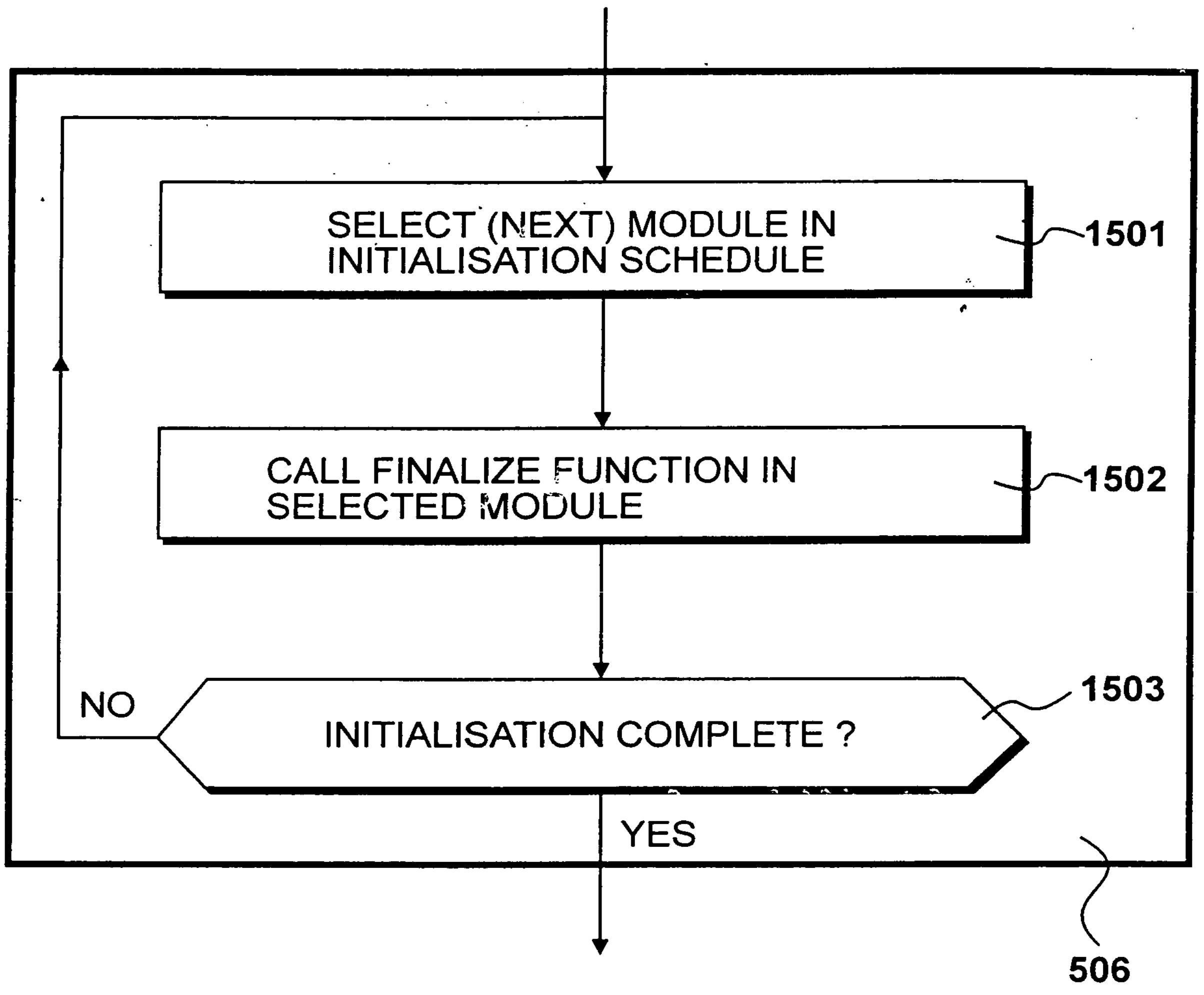

SELECT (NEXT) MODULE IN INITIALISATION SCHEDULE
1501
CALL FINALIZE FUNCTION IN SELECTED MODULE
1502
NO
INITIALISATION COMPLETE ?
1503
YES
506

Figure 15

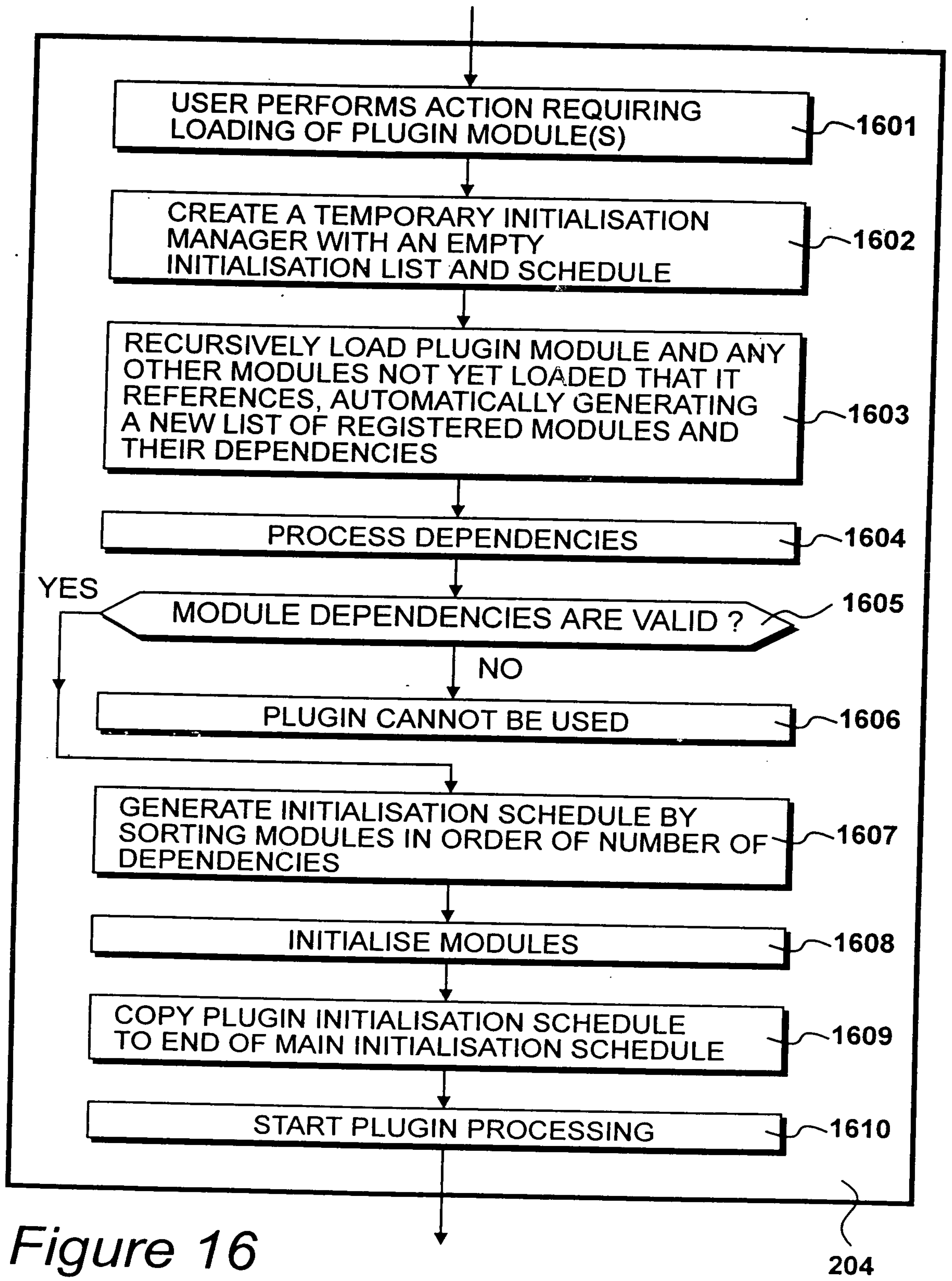

USER PERFORMS ACTION REQUIRING LOADING OF PLUGIN MODULE(S)
1601
CREATE A TEMPORARY INITIALISATION MANAGER WITH AN EMPTY INITIALISATION LIST AND SCHEDULE
1602
RECURSIVELY LOAD PLUGIN MODULE AND ANY OTHER MODULES NOT YET LOADED THAT IT REFERENCES, AUTOMATICALLY GENERATING A NEW LIST OF REGISTERED MODULES AND THEIR DEPENDENCIES
1603
PROCESS DEPENDENCIES
1604
YES
MODULE DEPENDENCIES ARE VALID ?
1605
NO
PLUGIN CANNOT BE USED
1606
GENERATE INITIALISATION SCHEDULE BY SORTING MODULES IN ORDER OF NUMBER OF DEPENDENCIES
1607
INITIALISE MODULES
1608
COPY PLUGIN INITIALISATION SCHEDULE TO END OF MAIN INITIALISATION SCHEDULE
1609
START PLUGIN PROCESSING
1610
204

Figure 16

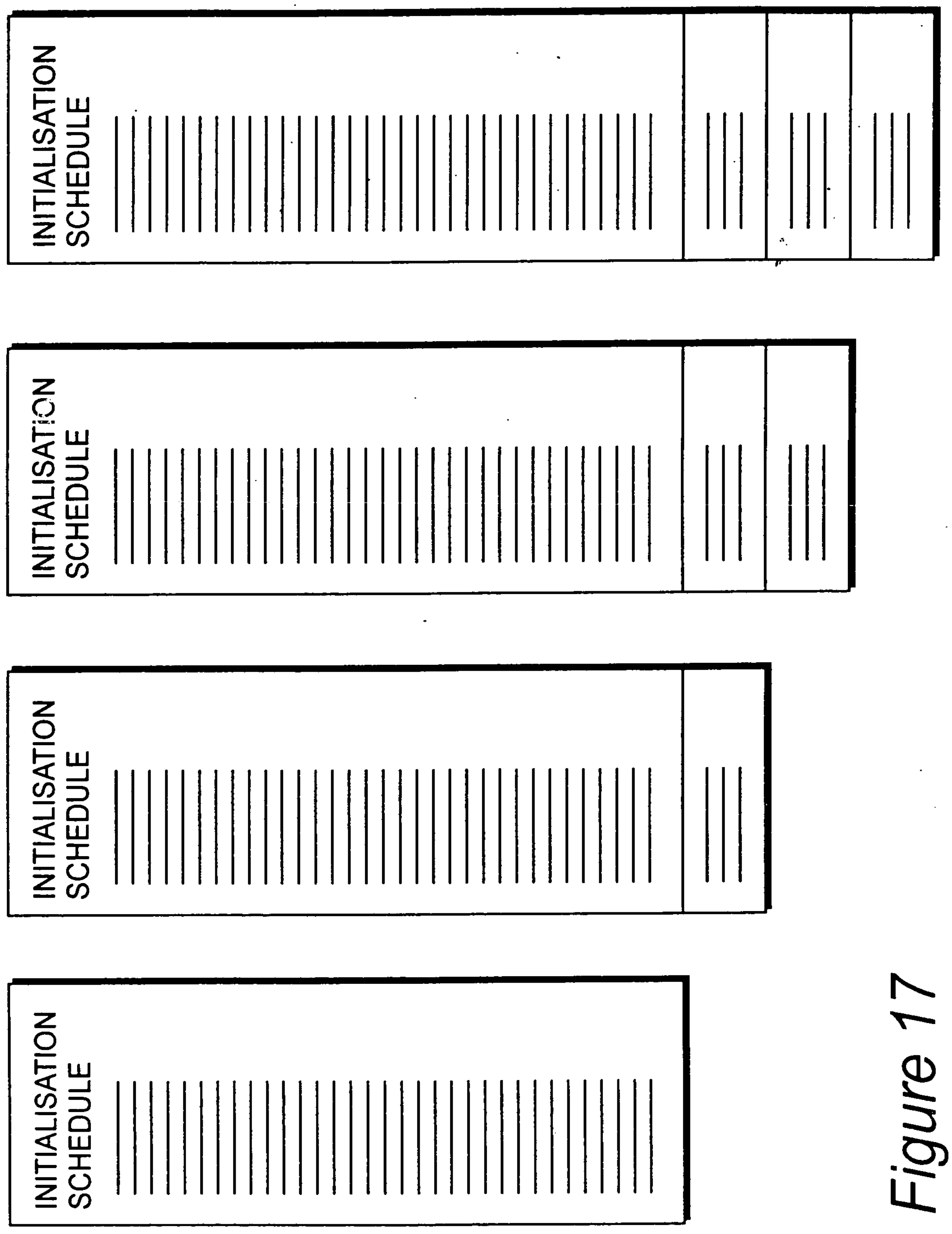

INITIALISATION SCHEDULE
INITIALISATION SCHEDULE
INITIALISATION SCHEDULE
INITIALISATION SCHEDULE

Figure 17

2034-P553-GB

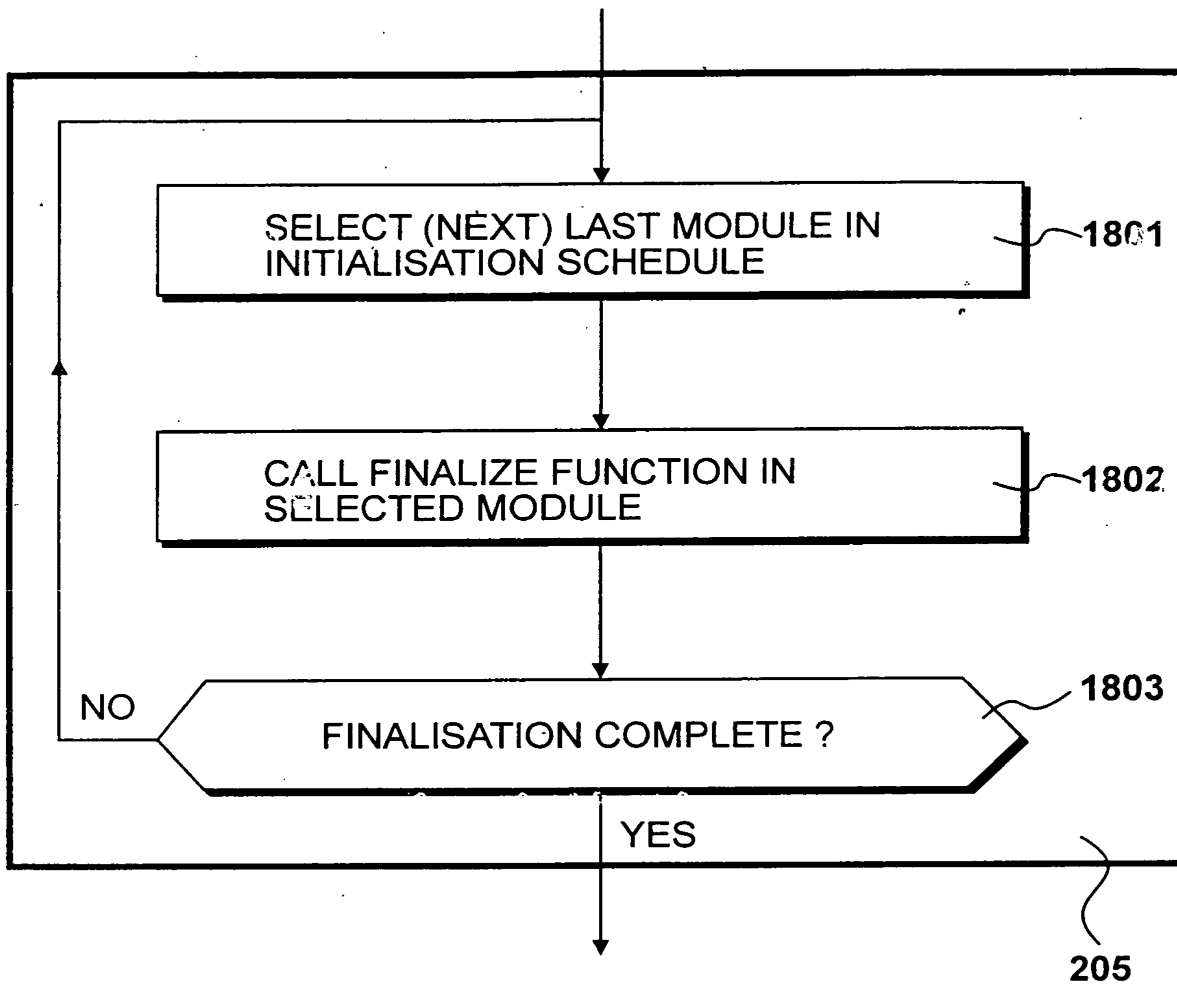
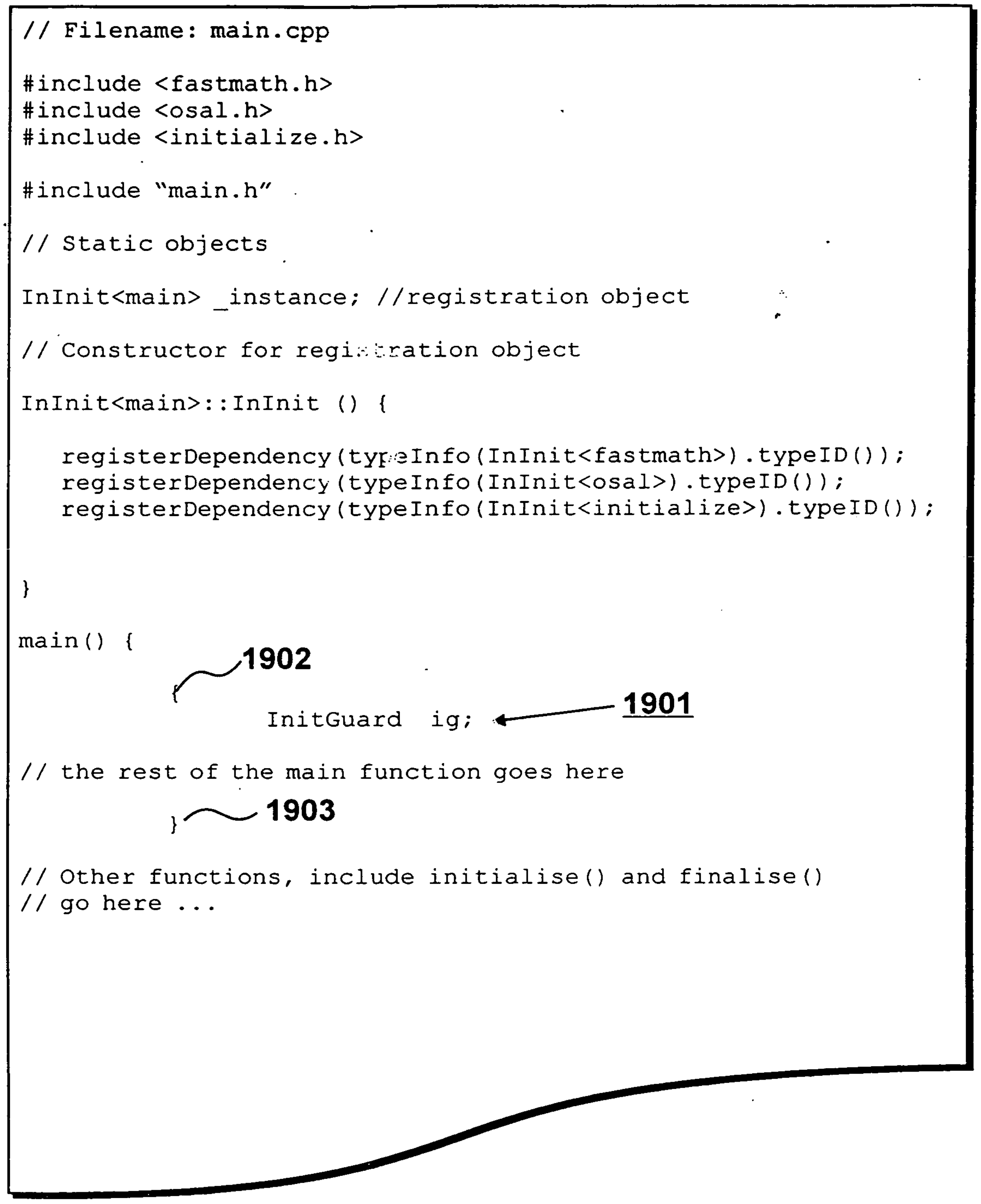
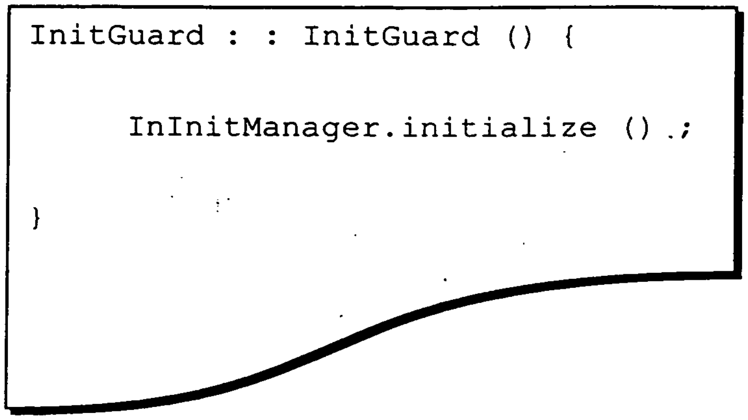

SELECT (NEXT) LAST MODULE IN INITIALISATION SCHEDULE
1801
CALL FINALIZE FUNCTION IN SELECTED MODULE
1802
NO
FINALISATION COMPLETE ?
1803
YES
205

Figure 18



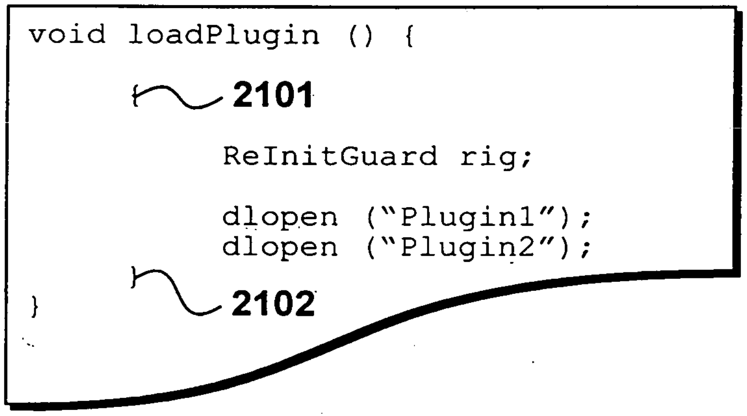
```
// Filename: main.cpp

#include <fastmath.h>
#include <osal.h>
#include <initialize.h>

#include "main.h"

// Static objects

InInit<main> _instance; //registration object

// Constructor for registration object

InInit<main>::InInit () {

   registerDependency(typeInfo(InInit<fastmath>).typeID());
   registerDependency(typeInfo(InInit<osal>).typeID());
   registerDependency(typeInfo(InInit<initialize>).typeID());


}

main() {
          {
                InitGuard  ig;

// the rest of the main function goes here
          }

// Other functions, include initialise() and finalise()
// go here ...
```

Figure 19



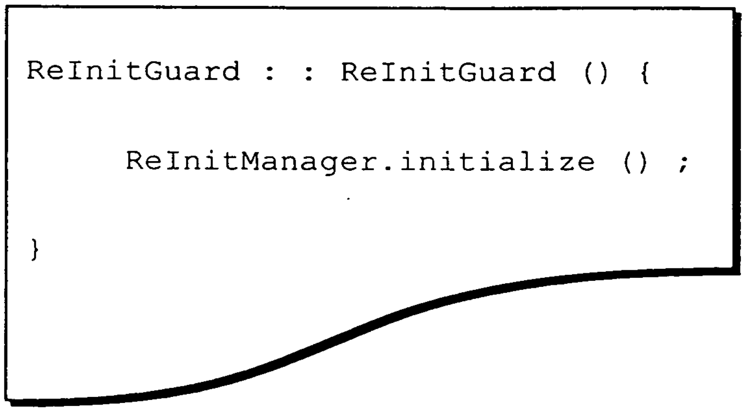
```
InitGuard : : InitGuard () {

     InInitManager.initialize () ;

}
```

Figure 20



```
void loadPlugin () {
     {
          ReInitGuard rig;

          dlopen ("Plugin1");
          dlopen ("Plugin2");
     }
}
```

Figure 21



```
ReInitGuard : : ReInitGuard () {

     ReInitManager.initialize () ;

}
```

Figure 22